ESD-TR-75-368

MTR-3105

THE HYBRID PROGRAM MEASUREMENT DEVICE: DESIGN AND CAPABILITIES

JANUARY 1976

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Bedford, Massachusetts

Project No. 572C
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract No. F19628-76-C-0001

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

Charles J. Hartman

CHARLES J. HARTMAN
Techniques Engineering Division

Henry J. Eiden

HENRY J. EIDEN, Major, USAF
Program Manager
Techniques Engineering Division

FOR THE COMMANDER

Frank J. Emma

FRANK J. EMMA, Colonel, USAF
Director, Information Systems
Technology Applications Office
Deputy for Command & Management Systems

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: ESD-TR-75-368	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): THE HYBRID PROGRAM MEASUREMENT DEVICE: DESIGN AND CAPABILITIES	5. TYPE OF REPORT & PERIOD COVERED
6. PERFORMING ORG. REPORT NUMBER: MTR-3105	
7. AUTHOR(s): D. A. Voorhies	8. CONTRACT OR GRANT NUMBER(s): F19628-76-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS: The MITRE Corporation, Box 208, Bedford, MA 01730	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: Project No. 572C
11. CONTROLLING OFFICE NAME AND ADDRESS: Deputy for Command and Management Systems, Electronic Systems Division, AFSC, Hanscom Air Force Base, Bedford, MA 01731	12. REPORT DATE: JANUARY 1976
	13. NUMBER OF PAGES: 93
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15e. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

PERFORMANCE MAXIMIZATION
SOFTWARE DEBUGGING
SOFTWARE OPTIMIZATION

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

External hardware may be employed to monitor an executing computer program at the instruction level without interfering with that execution in any way. Software may be used to control the external hardware, making the measurement process far simpler and easier to use. The Hybrid Program Measurement Device is useful for program debugging and optimization. Its capabilities and design are described in this MTR,

together with its implications and some recommendations for future work.

ACKNOWLEDGMENT

This report has been prepared by The MITRE Corporation under Project 572C. The contract is sponsored by the Electronic Systems Division, Air Force Systems Command, Hanscom Air Force Base, Massachusetts.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (concluded)

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION I

INTRODUCTION

The Hybrid Program Measurement Device (HPMD) is a programmable system which performs instruction-level monitoring of the execution of a minicomputer program. By combining both hardware and software, the HPMD provides debugging and optimization capabilities which are impossible to provide with hardware or software alone. The design and construction of the HPMD proved the feasibility of the hybrid hardware-software approach in providing an effective and powerful monitoring tool. The implementation of the device with a minicomputer not designed to be monitored in such a manner was quite difficult; discovering the nature and extent of such difficulties were important results of this effort.

Program measurement has traditionally been done by software tools alone; hardware has been used primarily for system-level measurements. With an HPMD, however, programmable hardware may be used to aid instruction-level debugging and optimization. In this context, "measurement" is meant in the broad sense of obtaining information which describes or defines the execution of a program.

The HPMD was implemented in conjunction with a Data General NOVA 800 minicomputer. The NOVA's memory and I/O buses are monitored using an associative memory to detect "events" such as the execution of a particular instruction or a data word being initialized. Many different events or combinations of events can be detected and used to control timing, counting, data storage, and breakpoint capabilities. Timing frequently used routines or the intervals between the occurrence of events can be valuable in optimizing execution speed or searching for a system problem which is time-dependent. The counters can be used to count the number of times a piece of code executes or the frequency of other specific events. Buffers and breakpoints allow access to otherwise obscure execution data which can be the basis for effective debugging.

The HPMD software runs before and after the user's program to initialize the hardware and to print the results. The software was written in ALGOL and assembly language, and it allows a straightforward yet more sophisticated human interface. Initialization is accomplished through interpretation of an interactive syntax designed specifically for this task. The results software performs data conversion and formatting chores to make the collected data more useful.

The scope of this document has been limited to a discussion of the nature of the design and what has been learned from its construction. The details of software code and hardware wiring are omitted as irrelevant to documenting what was done and what was learned as these may apply to future work. Instead, detailed functional descriptions are provided in the appendices which discuss the tasks and responsibilities of each software routine and hardware section. These functional descriptions, together with the working copy of the software listings and schematics, are adequate for extensions to the HPMD as well as its repair and maintenance.

The philosophy and considerations which went into the design are discussed first, since they focus the rationale behind the design itself. Next, the HPMD is described in terms of its appearance to an on-line user. This description defines the capabilities of the system and the syntax used to interact with the software. Brief summaries of both the software and the hardware then follow, which give overviews of the approaches used to implement the capabilities previously defined. The final sections discuss what has been learned from this effort and what conclusions can be drawn.

SECTION II

DESIGN PHILOSOPHY

The HPMD is intended as an application of the combination of hardware and software to instruction-level monitoring. It must be effective both as a debugging and optimization tool and as an experiment in hardware-assisted monitoring. Six criteria of effectiveness were used in its design.

First of all, the device must be easy to use. Time spent running the device itself is time directly added to the problem solution, and is frustrating and distracting for the user. The easier the HPMD is to use, the wider the range of problems to which it can properly be applied. More difficult devices are often considered "tools of last resort". A simple man-machine interface is needed to minimize procedure-related errors while setting up measurements or interpreting them. Such errors waste time or, if undiscovered, may ruin much work; they are also distracting to the problem-solver and tend to discourage further use of the device.

Secondly, the man-machine interface must be easy to learn; this is a fundamentally different requirement than to be easy to use. The interface must be clear and uncluttered, and there must be an easy-to-grasp relationship between commands and their effects. A minimum of unique formats and options helps, as does a resistance to catastrophe due to a mistyped character. Brief explanations must be available on demand to aid in learning and to clarify any misunderstandings.

Thirdly, to aid in learning and using the HPMD, the internal architecture of the device, with all its complexity and quirks, should be invisible to the user. In this way, the user can benefit from complex and powerful hardware yet need understand only the services it provides. This "black box" approach makes a hardware device appear much less imposing to a software-oriented programmer, and allows that programmer to focus his attention purely on the problem at hand.

Fourthly, the HPMD must not interfere with the operation of the computer it is monitoring. Conventional software debugging or performance measuring tools use the same computer resources as the program being measured. Unavoidably, the measurements interfere with the program operation, slowing it down significantly or aggravating system bottlenecks. Interference can be avoided if the HPMD does not share computer resources concurrently with the user's

program. The sharing of resources is a complicated process, and the exact nature of many types of interference is rarely known; certainty that some subtle effect is not distorting the measurements can only be achieved if there are no such effects in operation.

A non-interfering tool should be applied where interference would clearly render the results worthless. Critical timing relationships, a source of some of the most difficult systems problems, cannot be measured if the tool slows part of the system. Furthermore, the effectiveness of an interfering tool is curtailed if the speed of a lengthy user process is severly degraded. An interpretive execution scheme running programs at a hundredth of their original speed may be fine for short programs, but for finding once-a-day system bugs, it is useless. Only a tool which allows the system to run at full speed is effective in dealing with such an elusive problem.

The last two criteria of effectiveness for the HPMD are related to the experimental use of the HPMD in exploring hardware-assisted program measurement concepts. For this purpose, the HPMD should have generalized capabilities, rather than unique or obscure measurement capabilities with limited applicability. By flexibly applying general capabilities, a large number of meaningful instruction-level measurements can be available for evaluation. Exotic capabilities which are useable in only a few situations prove very little about the general concept.

Finally, if the results of such an experiment are to be widely applicable, the design must be easy to transfer to another machine or another environment. The less modification needed to transfer, the more general the solution. Where a device is totally dedicated to and dependent on the unique features of a particular computer, the knowledge gained from its use is unlikely to apply to different computers. If, on the other hand, the device monitors only those elements of a computer which are common to many designs, such as instruction addresses or I/O data, then whatever knowledge is gained should apply just as widely. Furthermore, if it is necessary to move the HPMD to another machine, the task is far less difficult; only a new interface is required rather than a redesign of the entire HPMD.

These six criteria: ease of use and of learning, invisibility of the architecture, lack of interference, generalized capabilities, and transferability, were important parts of the HPMD design philosophy. Complexity is also a major factor. The complexity needed to increase effectiveness also increases cost, errors, and countless problems in definition, application, and documentation for

such an effort. There is a delicate balance between effectiveness and complexity which must be considered in designing each feature of the HPMD hardware and software.

SECTION III

USER ENVIRONMENT

The physical equipment and logical operations with which the user interacts while making any program measurements can be defined as the user environment. Within this environment a programmer will use the HPMD to debug or optimize a computer program, and therefore, this user environment should be customized for these tasks as much as possible to make them efficient and effective.

There are two fundamental approaches to specifying the measurement of computer programs before execution begins. One is to save all values or occurrences of a particular event (each instruction, all I/O, all interrupts, etc.). Although this approach requires vast storage and its results can take a long time to peruse, it is an "open-minded" approach which makes no assumptions about what will be found. Alternatively, by making assumptions beforehand, the information gathered can be limited to that which is related to the suspected problem. Less information need be gathered and much less time will be needed to ferret out evidence of a program bug. These assumptions are less dangerous than many made during debugging because they are explicitly stated while specifying the measurements.

The first approach relies on powerful information reduction and storage capabilities; the second relies on flexible and powerful detection logic to resolve events of interest from all others. As hardware becomes less expensive, the first approach may be more useful for debugging. For all but the smallest programs, however, avoiding interference dictates using high-speed (equal to the CPU) storage media for saving data, and these media are currently quite expensive. The second approach, including both detection and reduction, is far less costly (though far more complex), and can handle a wider range of problems. It was chosen as the basic architectural approach for the HPMD. As a result, the user must provide considerable initialization information in order to specify the conditions under which measurements are to be made as well as the nature of the measurements themselves. It is for these interactions that the user environment is optimized.

Physically, the environment consists of a single workstation, from which all the operations necessary for debugging and optimization can be controlled. These include the editing, assembling, and execution of the user's program as well as the specification of HPMD measurements. Since the selective approach to

program measurement implies the entry of a large amount of information by the user, the use of the system console terminal is ideal. Its CRT and keyboard allow rapid interactions involving highly-detailed information. The console terminal, together with the line printer used for printing the measurement results and the front panel of the NOVA which is visible from the operator's chair, are useful for both the basic programming operations and for using the HPMD.

The logical operations necessary to control the HPMD can make all the difference in the usefulness and effectiveness of making measurements. The design philosophy dictates that such operations must be easy to learn and use, not require an understanding of the HPMD internal architecture, and allow generalized measurement capabilities. These qualities are embodied in a concise syntax for specifying the two basic functions of the device's measurements: event detection and information reduction. This syntax allows control of both the detection of important program events and the subsequent reduction of the execution information. The results software then can obtain and output the data gathered by this selective measurement process. The initialization syntax and results software (DEBUG and DISPLAY) are the two major HPMD programs which the user may run. Two other programs (RERUN and RESTORE) will be discussed later.

The selective approach is applicable to most debugging measurements. Clues from an earlier run usually point to a specific area in a program which must be investigated, and the programmer usually knows what to check first. The selective approach limits the measurements to information relevant to the problem, leaning heavily on the power and resolution of the detection logic.

Sophisticated detection is an unknown area, and if the HPMD is to be a worthwhile workshop for exploring such unknowns, it must have flexible and powerful detection logic. In the final design, more than half of the hardware and software is involved in specifying and performing detection functions. The subsequent reduction functions, although they actually gather the information, require little information to initialize and less hardware to implement.

DETECTION

The hardware detection logic of the HPMD compares instruction-level "parameters" with anticipated values, watching for an equality. The parameters (addresses, data, or device numbers -- see

Table I) are fundamental to program execution and frequently associated with bugs. Equality between the parameter and its anticipated values is the only relationship which can be detected; hence an associative memory can be used, greatly simplifying the hardware.

Table I

Parameters

Parameter	Mnemonic	Meaning
0	INS	Instruction
1	OPR	Operand
2	OPL	Operand if Read
3	OPS	Operand if Store
4	DI	I/O Data if Input
5	DO	I/O Data if Output
6	IA	Instruction Address
7	OPA	Operand Address
8	OAL	Operand Address if Load
9	OAS	Operand Address if Store
10	DCA	Data Channel Access Address
11	AST	Address if Any Store
12	INT	Interrupting Device #
13	DV#	I/O Device #
14	IN#	Input Device #
15	OU#	Output Device #

The detection comparisons are divided into eight groups, each associated with a parameter. Within each group, the selected parameter is compared with from one to four different anticipated values. If a parameter matches any of those values, the match constitutes a program "event" for that group. The same parameter can be compared in several different groups, but since the groups have varying reduction functions associated with them, this may or may not be equivalent to comparing a parameter with 8, 12, 16... etc., anticipated values.

The selection of any one of 16 parameters and its comparison with up to four numbers, repeated for each of 8 groups, provides an extremely flexible and versatile detection capability. Its power is increased by the ability to AND together different groups. The results of the four comparisons of group #1 can be ANDed, one by

one, with those of group #5. This double detection operation is possible in determining the events for groups #1-#4. Requiring two normal events to coincide before a program event is detected is excellent for detecting a particular variable (Operand Address if Store) becoming a particular value (Operand if Store), or a specific character, such as an EOF (I/O data if Input) being read from a given device (Device # if Input). By optionally combining the groups, the same (expensive) associative memory can be used in different modes to allow single or double comparisons. This doubling option can be selected on a group-by-group basis; for example, #1 and #5 can be ANDed, comparison by comparison, to detect group #1's program event, while groups #2, #3, #4, #6, #7, and #8 are run in the single comparison mode.

The parameters chosen are common to most computers based on a von Neumann architecture. Hence, the capabilities of the HPMD are generalized and lessons learned from it can be widely applied. The parameters are also simple and comprehensible for a programmer, since they are closely related to the actions of the program as well as to the machine. This relationship avoids the mental "switching-gears" which would be necessary if the HPMD forced the programmer to dwell on machine-level considerations during program debugging.

REDUCTION LOGIC

The HPMD provides five forms of reduction logic. A timer and several counters are controlled by events alone, two buffers save program parameters upon or between events, and the HPMD can issue an interrupt to halt the program upon any event. Each of the eight detection groups has assigned to it a "unique function" (timer, counters, buffers); alternatively, each can cause the interrupt. Thus if the timer and all the counters and buffers are used, there can be no halting interrupt. Conversely, if eight events are set to cause interrupts, no other measurements are possible. Combinations such as the timer and several interrupts are allowed. Note that detection using the doubling mode rules out the use of the "unique function" for the group paired with the one being doubled. The limits of various combinations of single and double detections driving unique functions and halts will become clearer as the assignment of the unique functions to each detection group is explained (see Table II).

Table II

Unique Reduction Functions For Each Group

Detection Group	Reduction Function
1	Start Timer
2	Stop Timer
3	Enable STORE Buffer
4	Disable STORE Buffer
5	Increment Counter #1
6	Increment Counter #2
7	Increment Counter #3
8	Trigger GRAB Buffer

Timer

The first two detection groups control the timer; group #1 turns it on, and group #2 turns it off. The timer, which can be repeatedly started and stopped to accumulate time, is a 30-bit counter driven at 5 MHz. A maximum time of 214.7483648 seconds can be measured with a resolution of 200 nsec., fine enough to notice the results of any change in the coding for the optimization of execution time. Longer periods can be measured, with the same precision, by determining the number of times that the 214.7483648 second counter overflows.

Counters

Groups #5, #6, and #7 each increment a counter upon detecting their program events. The counters are 20 bits long, providing maximum counts of 1,048,575 before wrapping around to zero.

Together, the timer and counters represent powerful tools for optimization. The counters can determine the frequency of execution of instructions, blocks of code, routines, or even whole programs. The timer can measure the time spent in various parts of a program or system. Together, they can determine the average execution time for any code or routine, by measuring the total time spent within it and the number of times it was executed. Hence, one can measure the changes in execution time resulting from different coding.

Buffers

The remaining detection groups control the two buffers within the HPMD (the STORE buffer and the GRAB buffer). These buffers implement two different approaches to saving program data; it was unclear which would be more useful, so both were included in the HPMD for evaluation. Having at least two buffers within any program measurement device is advantageous, since both addresses and data may be saved simultaneously. Relationships between the addresses and data can then be discerned.

STORE Buffer

The STORE buffer, which saves each consecutive value of a selected program parameter, is enabled and disabled by detection groups #3 and #4. The buffer itself holds 256 20-bit values. These 20 bits include a 16-bit parameter (as in Table I) and 4 extra bits. The extra bits tell whether the value saved originated with the CPU or the data channel (important for the AST parameter), whether it was a load/input or a store/output operation (important for OPA, DCA, OPR or DV#), and whether the GRAB buffer saved any information during the same memory cycle. These extra bits, under some circumstances, help the programmer identify the stored information. For example, if a location is being wiped out, it is necessary to know whether the CPU or the data channel is responsible.

The STORE buffer's operation can be modified by several control options specified by the programmer. The buffer can be frozen when it is full, preserving the first 256 values it received. Alternatively, the earlier values can be lost as more come in, making only the most recent 256 values available. When the buffer becomes full, an interrupt can be issued stopping the program and allowing the current 256 values to be printed; later, the program can resume capturing another 256 values, etc. (The interrupt capability will be more fully discussed later.)

Another option allows the buffer to be initially running or initially disabled. Finally, since the STORE buffer can be stopped and restarted repeatedly, an option exists to inhibit this restarting capability. The values of a parameter can be saved between the first two occurrences of an event, and subsequent passes between those events will not reenable the buffer. These four options allow the programmer to manipulate the gathering of information to best suit the application.

The STORE buffer has three major uses: historical, statistical, and detective. The historical use is the most

important; by saving consecutive values of a parameter, such as IA, a trace of the program behavior between particular events can be obtained. By saving 256 values of a parameter, a statistical data base can be built up, especially over several runs. For example, interrupt, data channel, or I/O activity can be observed and the percentages of that activity caused by the different devices can be discovered. Finally, examples of instructions which modify a core location or perform I/O can be used in tracking down a mysterious program bug.

GRAB Buffer

Whereas the STORE buffer saves every value of a parameter between two events, the GRAB buffer saves the value of certain parameters only upon detection of an event. Thus the programmer can determine the connection between a parameter and an event; if a location is altered, what value is stored into it?

The GRAB buffer also has several options which the programmer specifies when the measurements are selected. Like the STORE buffer, it can be frozen when full if desired, and an interrupt may also be initiated. Another option couples the GRAB buffer to the STORE buffer so that they are enabled and disabled by the same two events (detection groups #3 and #4). In this way, if the STORE buffer is enabled only during a particular block of code, the GRAB buffer will gather information only within that block. Hence the selectivity of the HPMD buffers is increased by limiting the scope of the measurement to only the particular area or areas of interest.

The information "grabbed" upon an event by the GRAB buffer is not the Table I parameters, as with the STORE buffer, because the relevant parameter values are often unavailable. Instead, the information saved by the GRAB buffer consists primarily of the "current" address, data, or device number. Table III shows the information available for each option and type of event. To provide more power for the programmer, the instruction's address (IA) is temporarily saved so as to provide a fourth option during an Execute cycle event. This allows, for example, the "grabbing" of the IA upon a subsequent operand store into a particular location, which results in the GRAB buffer filling with the address of each instruction which modifies a memory location.

Table III

Grab Buffer Input Information

Type of Host Computer Action	Grab Buffer Input Selection			
	Address	Data	Dev. #	IA
Instruction Fetch	IA	INS	Most Recent DV#	IA
Execute	OPA	OPR	Most Recent DV#	IA
I/O	IA	DI DO	DV#	IA
Data Channel	DCA	Data Trans-ferred	Most Recent DV#	Most Recent IA
Interrupt Acknowledge	IA	INS	INT	IA

It should be noted that some combinations of options and events are often meaningless, but since under some circumstances they may be useful, they are not prohibited. For example, "grabbing" the most recent device number upon a non-I/O event would usually be irrelevant.

By selecting the data option upon an event involving a specific DCA, the data channel data, a piece of information not otherwise available, may be grabbed. In designing the HPMD to be effective, this information was judged less useful in specifying events than the sixteen program parameters shown in Table I.

The GRAB buffer is useful for gathering historical and statistical data similar to that gathered by the STORE buffer. The basic difference is that the STORE buffer information is closely associated with a parameter alone, whereas the GRAB buffer reveals the relationship between a parameter and an event. It is, therefore, even more useful for uncovering examples of a mysterious occurrence; upon an event, the buffer can record the instruction,

address, or device number, etc., which indicates the cause or effect of the event itself.

Interrupts

The interrupt capability is a simple but powerful tool; upon an event, the program can be stopped. For this reason, any detection group not otherwise used can be allowed to trigger an interrupt, as can a full buffer. The interrupt itself is the same as from any I/O device with two differences: it cannot be masked out, and it works even though CPU interrupts are disabled.

The interrupt is very useful as a breakpoint; one may execute part of the program and then see if the error has occurred yet. Obscure bugs may be uncovered by subdividing the program into smaller and smaller pieces or a portion of the execution may be carefully examined. Breakpoints are basic tools whose usefulness have been proven in countless debugging systems.

By combining the interrupt capability with the other reduction functions, especially the buffers, the effectiveness of these other functions is greatly increased. Breakpoints simply allow examination of the current program state at the breakpoint; the interrupt and buffers combined allow the examination of recent program actions leading up to the breakpoint. This can be extremely useful if the event causing the interrupt is associated with the effect of some program bug. The cause of an event can often be revealed by displaying the program path or other behavior before that event.

As with most I/O interrupts, two or three instructions execute before the program stops. Only by designing the original CPU hardware or firmware to handle interrupts immediately could a true "halting" of the program be achieved without subsequent instructions being executed.

INITIALIZATION SYNTAX

There are four HPMD software programs which the user can run, but only DEBUG is used frequently. The DEBUG program is by far the largest and most complex part of the HPMD software, and is the only one which requests much input from the user. Its function is to specify measurements and carry them out. It therefore supports an interactive syntax which allows the initialization of the entire HPMD hardware while keeping the user environment simple and comprehensible.

The syntax commands can be thought of as divided into four catagories: specification, manipulation, actions, and assistance. The specification commands correspond to the reduction functions: TIME, COUNT, STORE, GRAB, and HALT. To manipulate the current measurement selections, DELETE, PRINT, ANNOTATE, KEEP, and USE are available. Two actions are possible: QUIT and RUN, and for assistance in using the syntax, line feed, "?", and "!" have special meanings.

A feature which is noticed immediately makes the syntax as quick and easy to use as possible: the user need type only the minimum information necessary to define his command or data uniquely. For example, only the first letter of each command must be typed by the user; the rest of the word or phrase is added immediately by the software. The result is a listing of the interaction which reads easily yet does not require much typing by the user. In the examples which follow later, the characters typed by the user are capitalized and underlined and those generated by the software are not.

Measurement Specifications

To describe the syntax, a few definitions are needed:

Parameter: any one of the sixteen aspects of program behavior listed in Table I.

Value: an unsigned octal number, 0-177777.

Event: the description of an equality between a parameter and from one to four values, with the values delimited by slashes.

Type: The type of information to be stored in the GRAB buffer, indicated by an "A","D","#", or "I" for Address, Data, Device Number, or Instruction Address, respectively.

Examples of events are: IA = 2000/2053/104 and DV# = 11. These events define a detection logic group's task, and are thus used in the syntax for each of the five measurement-specification commands. Events by themselves are sufficient to define the nature of the halt, timer, and counter measurements; the buffers must also be provided the type of information to save. The STORE buffer must be given a parameter to store, and the GRAB buffer is given the "type" of information. The formal definition and examples of each measurement specification follow.

Time from (EVENT)
until (EVENT)

(example: Time from IA = 401/407
until IA = 423)

Count if (EVENT)

(example: Count if IN# = 51)

Halt if (EVENT)

(example: Halt if OPS = 16503)

Store (PARAMETER) from (EVENT)
until (EVENT)

(example: Store OPR from IA = 4061
until IA = 4120)

Grab (TYPE) if (EVENT)

(example: Grab # if DI = 15)

These definitions and examples do not include the possibility of doubling the Time, Store, or Halt commands. If (EVENT) is expanded to mean either

"(PARAMETER) = (VALUE(s))"

or

"Both (PARAMETER) = (VALUE(s))

and (PARAMETER) = (VALUE(s))"

for those commands which allow doubling, then the previous command definitions are complete.

Doubling Examples:

Halt if Both DV# = 20/21
and DI = 42104/2347/7777

```
Store INS from Both OPR = 4
               and  OAS = 3047
         until Both OPR = 7
               and  OAS = 3047

Store IA  from   IA = 2031
          until Both IN# = 31
                and  DI = 15/12/177

Time from  Both   OPR = 177776/177777
           and    OAS = 3016
     until IA = 3070
```

Manipulation Commands

Manipulation of the measurement specifications which are being held within the initialization software is possible by using the Delete, Print, Annotate, Keep, or Use commands. Delete allows one to expunge from the current list of measurements to be performed any one or all of them. Simply typing the "D" for "Delete" and the first letter of the measurement ("T", "C", "S", "G", or "H") is all that is usually needed to eliminate that specification. If "A" is typed, all the current specifications are deleted. If the same type of measurement has been specified more than once, a number must also be entered to identify the one to be deleted. The specifications are numbered in the order of entry, which is also the order in which they are printed by a Print command or as part of the results printout.

The Print command asks for a "T" or "L" to indicate the terminal or the line printer. The Annotate command allows a large amount of description to be entered and saved with the current measurement specification. Any previous annotation is replaced. Lines of input may be typed as desired, terminated by two consecutive carriage returns. Keep and Use save and recall the current measurement specifications on the disk, allowing several sets of specifications to be maintained simultaneously. These commands require a file name to indicate where to put or find the measurements.

Action Commands

Two of the commands, Quit and Run, end the execution of DEBUG. QUIT forces the current measurement specifications, if any, into a disk file for reloading the next time DEBUG is entered; it is useful for building up and saving (with "Keep") several sets of specifications without using any immediately. Run loads the HPMD hardware with the current measurement specifications and then executes the user's program. These specifications are also saved on disk, as with the Quit command. The Run command therefore starts the measurement process with the hardware.

User Assistance

At any point in the syntax, typing "?" obtains a concise explanation of what information is desired by the software. Typing "!" obtains a listing of valid next characters to continue the current measurement specification; the "!" will be discussed in more detail later. A line feed aborts the current measurement specification and a "....NOT DONE" is echoed. No matter where one is in entering a command, if the command is not complete, a line feed insures it is not added to the current measurements.

The syntax is a concise means of entering complex information and relationships. To keep it as easy to learn and use as possible, a feature was added which in effect eliminates the possibility of syntax errors. At any point during the interactions between the user and the HPMD software, only a limited number of options for the user exist. The next character which may be typed in most contexts is usually one of ten or fewer. Characters whose entry would be "illegal" (meaningless in the context of the previous characters) are not accepted into the command and are not echoed to the user. Therefore, typing syntax errors is impossible, and no error messages are ever required.

This interaction with the keyboard is a dramatic change for the user first experiencing it, but is not as frustrating or confusing as one might expect. Users learn a syntax new to them chiefly by trial and error. With the type of interaction described above, the user gets immediate and non-destructive feedback from the software, because an illegal character, whether caused by a misunderstanding or a slip of the finger, is not echoed on the terminal. Furthermore, the previous characters remain undisturbed, so the line need not be reentered, reducing frustration and making the trial and error learning process faster and less painful.

The usefulness of the "!" command now becomes clear; the characters listed when a "!" is entered correspond to the "legal" keys. The "!","?", and line feed are always legal. The "!" can be used to resolve any confusion as to the demands of the syntax at each point. The "?" provides a fuller explanation, to both teach the user having little experience, and to remind more experienced users of such things as a parameter mnemonic.

PROGRAM EXECUTION

All these commands are handled by DEBUG, which is executed from the console. To measure a particular program (via the RUN command), the program's name must be typed after the "DEBUG", complete with all the parameters normally used when executing the program. For example, to measure the assembler which is itself assembling another program named XYZ, the programmer might type "DEBUG ASM XYZ".

Results Software

The results software runs under three circumstances: upon an interrupt, upon completion of the user program, and upon direct command by the programmer. Its purpose is first to read in the time, counts, and buffers' contents from the HPMD, and then to convert, format, and list them.

The results printout contains many types of information relevant to the particular measurement (see Figure 1). First is a distinctive title and pattern at the top of the page which makes it easy to discern from all other line printer output. The time and day are listed, based on the system's internal clock; since the time is given to the second, and the results software takes more than a second to run, the time and day are a unique identifier for each run. The annotation is then printed, if one was entered with the measurement specifications. The annotation helps identify the printout in terms of the programmer's purpose in making the measurement. Next, the specifications of the measurements themselves are listed. This is the same listing that the initialization software provides if a "Print measurements at Lineprinter" command is given.

These pieces of information, although not gathered by the HPMD hardware, are most useful in keeping track of that data. It is a disadvantage of computer printouts that they all look alike: the numbers may vary, but formats frequently remain the same. Time, date, comments, and specifications all help distinguish each results printout.

```
                    ○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
                    ○ PROGRAM MEASUREMENT DEVICE ○
                    ○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○


    02:51:14                                              01/30/75


COUNT TIMES EXECUTED (ONE).

TIME            FROM    IA  = 000436 / 000436 / 000436 / 000436
                UNTIL   IA  = 000437 / 000437 / 000437 / 000437
COUNT           IF      IA  = 000436 / 000436 / 000436 / 000436
STORE   IA      FROM    IA  = 000453 / 000453 / 000453 / 000453
                UNTIL   IA  = 000456 / 000456 / 000456 / 000456
GRAB    A       IF      IA  = 000453 / 000454 / 000455 / 000456


          TIME  = 210.5464400 SECONDS

          COUNT = 1046513


 STORE BUFFER:

130011+SD 130011+SD 130011+SD 130011+8D 130011+SD 130011+8D 130011+SD 130011+SD
130011+8D 130011+8D 130011+SD 130011+SD 130011+8D 130011+SD 130011+SD 130011+SD
130011+8D 130011+8D 130011+SD 130011+SD 130011+SD 130011+SD 130011+SD 130011+SD
130011+8D 130011+SD 130011+8D 130011+SD 130011+SD 130011+SD 130011+SD 130011+8D
130011+8D 130011+8D 130011+SD 130011+8D 130011+SD 130011+SD 130011+SD 130011+SD
130011+8D 130011+SD 130011+8D 130011+8D 130011+8D 130011+SD 130011+8D 130011+SD
130011+8D 130011+8D 130011+SD 130011+SD 130011+8D 130011+SD 130011+SD 130011+SD
130011+8D 130011+SD 130011+SD 130011+SD 130011+SD 130011+SD 130011+SD 130011+SD
130011+8D 130011+8D 130011+8D 130011+SD 130011+8D 130011+8D 130011+SD 130011+SD
130011+8D 130011+8D 130011+SD 130011+SD 130011+SD 130011+SD 130011+SD 130011+SD
130011+8D 130011+8D 130011+SD 130011+8D 130011+SD 130011+SD 130011+SD 130011+SD
130011+8D 130011+SD 130011+SD 130011+8D 130011+SD 130011+SD 130011+SD 130011+SD
130011+8D 130011+SD 130011+SD 130011+8D 130011+8D 130011+SD 130011+SD 130011+SD
130011+8D 130011+SD 130011+8D 130011+SD 130011+SD 130011+8D 130011+SD 130011+SD
130011+8D 130011+8D 130011+SD 130011+SD 130011+SD 130011+SD 130011+8D 130011+SD
130011+SD 130011+8D 130011+SD 130011+8D 130011+SD 130011+SD 130011+SD 130011+8D
130011+8D 130011+8D 130011+8D 130011+SD 130011+8D 130011+SD 130011+SD 130011+8D
130011+8D 130011+8D 130011+SD 130011+SD 130011+SD 130011+SD 130011+SD 130011+SD
130011+8D 130011+8D 130011+SD 130011+8D 130011+8D 130011+SD 130011+SD 130011+8D
130011+8D 130011+8D 130011+SD 130011+8D 130011+SD 130011+SD 130011+SD 130011+SD
130011+8D 130011+SD 130011+SD 130011+SD 130011+SD 130011+SD 130011+SD 130011+SD
130011+8D 130011+8D 130011+SD 130011+SD 130011+SD 130011+SD 130011+SD 130011+SD
130011+8D 130011+8D 130011+8D 130011+SD 130011+SD 130011+SD 130011+SD 130011+SD
130011+8D 130011+SD 130011+SD 130011+8D 130011+SD 130011+8D 130011+SD 130011+8D
130011+8D 130011+8D 130011+8D 130011+SD 130011+SD 130011+8D 130011+SD 130011+8D
130011+8D 130011+8D 130011+8D 130011+SD 130011+SD 130011+SD 130011+SD 130011+8D
130011+8D 130011+8D 130011+8D 130011+SD 130011+SD 130011+SD 130011+SD 130011+8D
```

Figure 1. Sample Results Printout

The remainder of the printout provides the data gathered. Each type of measurement is shown only if specified. For example, if only two counters and the GRAB buffer were specified by the programmer using the initialization software, then no mention is made in the results printout of the timer, the third counter, or the STORE buffer. This makes the printout shorter and simpler, and clarifies the difference between a zero time and an unused timer, or an empty buffer and an unused one. The counters are not numbered, either in the specifications or in the results; the order of the counters is assumed to be the same for each.

The time measurement is a 10 digit decimal number labelled as seconds. The counts are 7-digit decimal numbers. The buffers are listed as arrays of 256 entries, in 32 rows of 8 columns. Each entry consists of the value saved, given as a 6-digit unsigned octal number, followed by three modifying characters. The first may be a "+": if present, it means the other buffer saved a value on the same memory cycle. This indication can be very useful in relating the information in one buffer with that in the other. Next may be an "S" indicating the direction of information flow. If present, it indicates that the value was transferred out of the CPU, as in a Store operation or a Data-Out instruction. If there is no "S", then the information flowed into the CPU, as in a Read or Data-In instruction. This is useful when GRABbing the IA upon the accessing of a location (OPA). All instructions which reference that location would be listed, with the "S" indicating which stored into it. Lastly, a "D" may be shown to indicate that this value came from the data channel rather than the CPU. When searching for the reason a table in core is being overwritten, the AST parameter, if stored, will show the actual overwriting. The "D" character will then reveal whether the data channel or the CPU was responsible.

User-Run Programs

The HPMD software includes four programs which can be run by the user. The largest and most important, "DEBUG", has already been discussed. Next largest is the results software, which may be run by typing "DISPLAY". Since it is run automatically when the program completes or reaches a breakpoint, the user will not normally need to run it manually. If, however, the program crashes the system or must be cancelled by the user through the system console, the "DISPLAY" program can be run to print the measurements on the line printer.

"RERUN", as its name implies, reloads the HPMD hardware and then executes the user program again. This is a shortcut for reentering "DEBUG" and issuing the "RUN" command. It saves time

when the same program is run repeatedly with the identical measurements each time, which is often the case while a program is being tested.

The final program the user can run restores NOVA interrupt processing to normal if the user program is aborted. To implement breakpoints, the system interrupt vector is altered for the duration of the user's program. When the user's program completes, the altered vector is restored. Restoration must be performed manually by running RESTORE if there is an abnormal completion.

SECTION IV

SOFTWARE

The HPMD software performs four key functions: obtaining the measurement specifications, starting the actual measurements, handling breakpoints, and printing the results. These functions are implemented by five programs: DEBUG, RERUN, RESTORE, and DISPLAY which are "visible" to the user, and QZ, which is not. Upon specifying the "Run" command in DEBUG, the remaining four programs are automatically executed to handle the measurement process. As such, they could be considered parts of a single, large program, but their separation allows them to be invoked manually to provide the convenience of "RERUN" and the recovery capabilities of "RESTORE" and "DISPLAY".

Few aspects of the software do not manifest themselves at the user environment level. Therefore, a simplified functional description of how the HPMD programs affect the user environment has already been presented. The implementation of these functions was a straightforward task, and although the initialization software is quite large, at no point does the HPMD software become overly complex. Major difficulties were encountered with the systems programming to allow interrupt handling, but these were due to the necessity of exploring countless schemes which used or misused the operating system. Due to a lack of documentation in several areas, trial and error experimentation was needed to determine flaws in many simple approaches. Thus the final scheme is not particularly complex despite requiring a great deal of time to develop.

INITIALIZATION SOFTWARE

DEBUG is an interactive syntax interpreter which also prepares the resulting measurement specifications for loading into the hardware by RERUN. The two programs communicate via a disk file, $HPMD.MS, where DEBUG puts the specifications, annotation, and user program name. That program name is then read by RERUN and executed, since DEBUG has already performed the bulk of the initialization process.

The primary purpose of the initialization software is to convert the measurement specifications from the user syntax to the radically different hardware-loadable table. A simple command, such as the user typing "H", "IA", and "2047", is expanded into "HALT

PROGRAM IF IA = 2047", providing a readable record. Only after all the commands are entered can a detection group be assigned to handle this command. The assignment includes four associative memory locations (all filled with 2047), a corresponding detection parameter of 6 (IA), and assorted bits specifying halting (as opposed to the normal group reduction function) and a lack of doubling. It is in the difference between these two forms for the same information that much of the power of the HPMD concept lies; the conversion between a human-oriented syntax and a hardware-oriented data format allows each to function efficiently.

In this way, the user never need deal with the hardware architecture. The measurements are accumulated without being assigned specific hardware detection groups, since that assignment should not be done until all the specifications are obtained. Since the count and halt measurments may be performed by any of several detection groups, the premature dedication of a specific group to a particular measurement might rule out a subsequently-specified measurement which depends on that group. Upon a "Run" command, all the assignments are performed, and a hardware-loadable array is built up and saved in $HPMD.MS. This juggling of information is a major feature of the initialization software.

Rarely were tradeoffs made in favor of the software at the expense of syntax tidiness. DEBUG is by far the largest HPMD program (1240 lines of ALGOL), and size was never considered a factor in its design. The 26 routines include 7 shared subprograms; this high degree of sharing significantly reduces the amount of code; if that code were in-line rather than in subprograms, another 1740 lines of ALGOL would have been added. The initialization syntax is relatively easily adaptable to software interpretation although requiring quite a bit of code. Features such as the availability of a line feed/abort in every context and the entry of only the minimum necessary characters demanded a good deal of software, but their contribution to the syntax makes that additional software worthwhile.

The architecture of DEBUG closely follows the syntax it interprets. The syntax consists of two kinds of commands, those which specify measurements (Count, Grab, Time, Store, and Halt), and those which affect those specifications (Delete, Print, Annotate, Run, Keep, and Use). For each of these 11 commands there is a separate routine which processes that command. The measurement specifying routines are in turn supported by other routines which fetch parameters and values, check if doubling is possible, and ask the user about the buffer control options. The remaining command routines are supported by routines which format a measurement

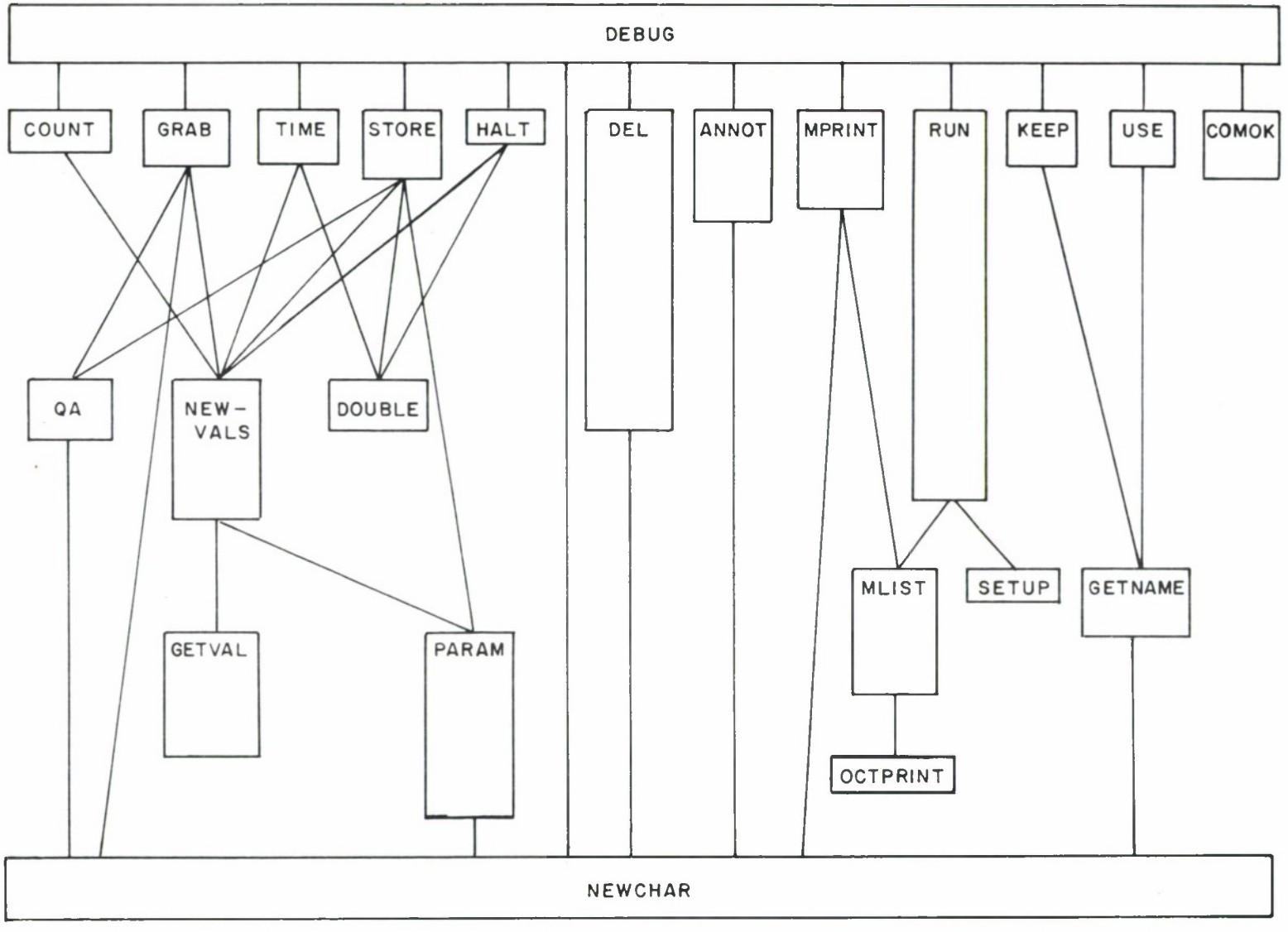

Figure 2 INITIALIZATION SOFTWARE ARCHITECTURE

listing, handle disk file names, and execute the user program. Many of the routines on each level are supported by the basic keyboard management routine. This architecture is diagrammed in Figure 2.

EXECUTION SOFTWARE

The execution software has two distinct responsibilities: loading the hardware at the beginning of the user's program, and processing the HPMD interrupts. The former task is quite straightforward, in that the initialization software has already done most of the difficult work by creating the TABLE array in ready-to-load format. The processing of interrupts, however, is the single most difficult software task. Its implementation is shared by part of RERUN and all of QZ and RESTORE.

RERUN

RERUN is in a sense the "main-line" of the execution software, in that it swaps to the user program, RESTORE, QZ, and DISPLAY (the results software). By making each of these a separate program, the user can run them independently if needed. RERUN executes the user's program, and handles the actions immediately preceding and following that execution. It loads the HPMD hardware with the measurements specified by the initialization software, puts an interrupt servicing routine (ISR) into core if needed, swaps in the user program (which runs to completion), shuts down the HPMD hardware, and then runs RESTORE and DISPLAY. Since this sequence can be repeated, "RERUN" was chosen as the name to imply for the user that this program handles the entire instrumented execution.

The loading of the hardware is a simple process because DEBUG has produced an array of measurement specifications already transformed into a hardware-loadable format, so the hardware can be loaded with this array by a sequence of 57 data transfers. Since the HPMD hardware lacks the mechanical delays which are part of most peripheral devices, it can accept the data as fast as the NOVA can send it, requiring no busy flags or ready bits.

Interrupt Handling

Handling HPMD interrupts during the execution of the user program is difficult because the operating system cancels interrupt definitions upon completion of the defining program. Furthermore, all user interrupt definitions are removed upon a transfer to another program by chaining or swapping. Thus, if HPMD interrupts were to be processed by the operating system during the user

program's execution, the user program itself would have to make that definition. This is clearly an unreasonable burden for the programmer. Furthermore, the interrupt servicing routines (ISR) which the operating system does allow to be defined are prohibited from swapping in other software, or even doing disk I/O. Since the results software is relatively large, and the ISR must be in core with the user program, the ISR must be limited to code which swaps in the larger results software. This conflict ruled out using the operating system to handle HPMD interrupts.

To bypass the operating system interrupt restrictions, the user program must appear to be requesting the results software program swap. To do this, RERUN copies an ISR into high core beyond the user program and alters the interrupt vector to reroute all interrupts through that ISR. The ISR first identifies whether the interrupt is from the HPMD or not, and if not, branches to the operating-system interrupt-handling logic. If the interrupt is from the HPMD, the QZ program is run via a program swap. QZ quizzes the user as to the disposition of this breakpoint: continue, restart, or quit. The DISPLAY program is then run via a swap, and then the requested disposition is carried out. All these functions appear to the operating system as if they were invoked by the user program itself, since an HPMD interrupt is invisible to the operating system software, which is not informed of even the existence of the HPMD interrupt. It is clear that such a deception involving the interrupt vector depends on the ISR, which is not protected by the operating system, remaining intact. The RESTORE program restores the original vector and thus eliminates that dependence as soon as the user program completes.

RESULTS SOFTWARE

DISPLAY is a program which handles the entire task of producing the results printout. The printout contains information of three types: the date and time of day, a listing of the measurements with the annotation, and finally the data collected by the HPMD hardware. The time, date, and annotation are first printed; the time and date are available from the operating system via a system call, and the annotation is saved in $HPMD.MS. Input instructions obtain 10-bit values directly from the hardware, and conversion routines can then assemble decimal or octal numbers from several of these values taken together. The timer and each of the counters have their results displayed in decimal, whereas the buffer contents are given in octal, since quite often they are machine address or device numbers. By placing the conversion burden on the results software, the hardware can use an efficient binary format while the user sees the

more useful decimal or octal equivalent. The results software itself is written entirely in NOVA assembly language, to allow easy bit-level manipulation of the data. It was also discovered that the NOVA ALGOL multiprecision arithmetic was bug-ridden and therefore unsuitable for dealing with the high-precision timer and counter values. DISPLAY's modular architecture consists of 14 routines totalling 656 source lines and assembling into 1460 words of core. Much effort was expended in writing the results software to minimize core usage, since it was unclear at that time whether any system's programming scheme for handling interrupts could be found which did not force all of DISPLAY to be resident in core with the user's program. For this reason, in some cases the routines were delineated not by the rules of functional separation but rather to minimize core by eliminating almost all duplicate coding.

SUMMARY

Software is valuable in providing tools for the management of the measurement specifications and the presentation of the collected data. The KEEP, USE, and PRINT commands help the user handle the specifications as a group, and the DELETE and ANNOTATE commands modify that group. The user can then deal with debugging and optimization experiments in a very straightforward way; only the program, a group of measurement specifications, and the results need be considered. This simplified framework is cleaner and more conducive to the scientific method than schemes which involve the details of measurement implementation.

SECTION V

HARDWARE

The HPMD hardware carries out the measurements specified by the user with the initialization software. With access to every memory and I/O transfer, the hardware performs the monitoring in parallel with the user program's execution, and in no way interferes with the host computer.

The overall hardware design consists of two parts physically separated but connected by a cable. The information flowing over the cable can be defined conceptually in terms which are independent of the specific machine being monitored. The HPMD consists of a programmable program measurement device connected to an interface for the specific computer being monitored and controlling software which runs on the computer before and after the measurements. The division into two parts allows the HPMD to approach its design goal of transferability. To move it to a different 16-bit minicomputer, a new interface must be designed and built and the ALGOL software recompiled. The assembly language code must be partly translated and, to the extent it is operating system dependent, rewritten. Since the HPMD and the monitored minicomputer are deeply intertwined at many levels, it was impossible to make the HPMD any more machine-independent than this; all that is universally applicable is transferable, all that is machine or operating-system dependent is not.

The event detection and information reduction sections perform the measurements specified by the user. To allow the software to control these sections and to read back the results, the peripheral interface circuitry adds another path between the CPU and the HPMD.

Table IV summarizes the size and complexity of these three major sections plus the front panel logic. Physical size and wiring complexity are proportional to the chip counts, with an average of 15.8 pins per chip. The functional complexity, however, varies with the gate count. The storage buffers in the information reduction section account for a large number of the gates, but since they are implemented using LSI shift registers, little space or wiring was needed.

Table IV

Chip and Gate Counts

	Detection	Reduction	Device I/O	Front Panel	Total
SSI	108 chips	42 chips	39 chips	16 chips	205 chips
	538 gates	243 gates	179 gates	88 gates	1048 gates
MSI	97 chips	47 chips	38 chips	4 chips	186 chips
	1849 gates	1619 gates	1621 gates	94 gates	5255 gates
LSI	32 chips	20 chips	-none-	-none-	52 chips
	4416 gates	21080 gates	-none-	-none-	25496 gates
TOTALS					
	237 chips	109 chips	77 chips	20 chips	443 chips
	6803 gates	23014 gates	1800 gates	182 gates	31799 gates

The detection logic is what makes the HPMD different from other measurement devices. Systems-level monitors have large information reduction capabilities, often including built-in tape drives and extensive post-processing of measurement data. They do not allow detection of multiple program events similar to that defined in Section III. Using the initialization syntax, it is possible to request up to 32 simultaneous comparisons be performed, involving 8 different program parameters. These 32 simultaneous comparisons represent a powerful detection capability which permits using the selective approach to debugging and optimization.

DETECTION

Event detection may be conceptually thought of as a four-phase process. First, all sixteen parameters listed in Table I are synthesized from the host computer bus signals. Second, for any particular event, the parameter involved is selected from among the sixteen available. Third, the selected parameter is fed into a section of the associative memory, where it is compared against four preset values. Any resulting "match" signal then goes to the final detection section, the doubling logic. Here it may be gated with

other "match" signals to produce an "event" signal, thus completing the detection process.

In reality, the process is slightly different. None of the parameters is updated each memory cycle; even Instruction and Instruction Address are available only once per machine instruction. Others, such as the I/O parameters, may be seen only rarely. Also, the parameters involve only three "types" of information: addresses, data, and device numbers. Every memory cycle will have both an address and a word of data, and if programmed I/O is performed, a new device number will be available. The HPMD detection logic is based on these "types" of information rather than each parameter separately, and therefore much of the logic deals only with three sources of information, not sixteen. By sampling under the appropriate conditions, specific categories within each type are selected -- the parameters are identified in Table I. For example, by sampling the address bus only on non-data-channel FETCH cycles, only the instruction addresses are selected.

To detect parameters, it is therefore necessary for the HPMD hardware to obtain the current address, data, and device number, as well as a description of the kind of memory cycle. These form the machine-independent information which flows from the interface over the cable to the rest of the HPMD hardware. Five descriptor bits characterize the current memory cycle to allow identification of the parameters from the "types" of information available. The interface synthesizes these bits from the specific machine operations. These bits define the direction of information flow, the cycle initiator, and the purpose of the cycle.

The descriptor bits are used to synthesize sixteen "control pulses", one for each parameter. Each control pulse indicates when the corresponding parameter is updated. These pulses, combined with the address, data, and device number, constitute all the information necessary to synthesize each parameter.

The parameters are not directly synthesized, however, for a very simple reason: if each of the eight detection groups had to select between sixteen "parameter" registers plus the front panel switches (for testing purposes), the multiplexer required would be immense.

Instead, the Information Multiplexer selects between the three "types" plus the front panel switches, and therefore it can use dual 4-input multiplexer chips. The control pulses are gated with the associative memory output rather than the "type" information. The associative memory compares its values with every value of the

selected "type", not the desired parameter, but the irrelevant comparisons are not sampled by the control pulse, and so the result is the same. Even using this shortcut, the Information Multiplexer is the largest single hardware section, requiring 80 16-pin chips.

The detection process requires less than a microsecond, with most of that time devoted to obtaining the host computer bus information. The Information Multiplexer and associative memory themselves require less than 60 nsec. to select and compare the incoming data. This fast yet detailed examination of the program execution is used to control the information reduction functions.

REDUCTION FUNCTIONS

Detected program events control five types of reduction functions: timing, counting, continuous information storage, intermittent information storage, and program halting. These of course correspond to the Time, Count, Store, Grab, and Halt command defined in Section III.

A major objective of the HPMD experiment was the exploration of the selective approach to program measurement. An advantage of the selective approach is that general, straightforward reduction functions are sufficient. If the detection operations can resolve the precise events of interest, a very simple recording mechanism can capture the needed information.

The timer is a 30-bit binary counter driven asynchronously at 5 MHz. This 5 MHz. pulse train is gated by a set-reset flip-flop, which itself is driven by the start timer and stop timer events.

Each of the three counters is 20 bits long, and is implemented using MSI 4-bit binary counter chips, as is the timer. Both the timer and the counters are reset by an NIOS 5 instruction on the NOVA, which is fed to the reset input on each counter chip. This instruction is executed just before each measurement run.

Both the STORE and the GRAB buffers are built using LSI dual 256-position shift register chips. Since each small (10-pin) chip holds 512 bits, only ten are needed for each buffer. Thus these twenty chips contain more than half of the gates in the entire HPMD hardware (see Table IV).

The STORE buffer, like the timer, is controlled by a set-reset flip-flop driven by a pair of detection events. Its input can be

any of the parameters listed in Table I. The Information Multiplexer section provides "type" information to a 16-bit latch which is triggered by the STORE buffer control pulse. This same control pulse also shifts the entire buffer whenever a new parameter value is obtained. Extra counters and gating implement the buffer options such as permanent shutdown when full. Four extra bits are synthesized and stored along with the selected parameter. They indicate that the information is valid, that the GRAB buffer is also shifting, as well as the information in two descriptor bits.

The GRAB buffer differs from the STORE buffer mainly in its input. Instead of capturing each new value of a parameter while enabled, the GRAB buffer event triggers a buffer shift directly. Its input is the address, data, or device number plus a latch which provides the IA. In most other ways, it is identical to the STORE buffer.

The breakpoint capability is quite complex despite its straightforward function. First, a "HALT" signal is derived from the 11 possible causes within the HPMD (8 detection groups, 2 buffers and a front panel button). This signal is then sent over the cable to the interrupt generator, a part of the HPMD-NOVA interface. If HPMD interrupts are enabled (a manual and programmable function), the interface uses the conventional I/O device protocol to initiate an interrupt. The NOVA CPU is also forced into the Interrupt Enable state by driving a CPU signal, EXT ION EN. The original state is meanwhile preserved within the interrupt generator, and may be read out by the software for later restoration.

The ideal breakpoint is impossible to implement using an I/O interrupt; their purposes are dissimilar. A breakpoint is in theory unmaskable and immediate, without any destruction of the current machine state. An I/O interrupt is simply a method for getting the CPU's attention without requiring frequent busy-bit sampling. The HPMD interrupt successfully bypasses all masking or interrupt-disabling obstacles, but several instructions execute before the ISR begins, effecting a form of "skidding" beyond the event detected. Without redesigning the CPU, this fault is unavoidable.

PERIPHERAL INTERFACE

The peripheral interface allows the HPMD hardware to be programmed under software control, and its results read back and processed by other software running on the host computer. The loader directs and stores the controlling measurement information,

and the reading circuitry allows the reduction functions' data to be read back into the NOVA 800.

The loader is based on a decoded 6-bit counter which directs each successive data transfer to a different storage element. Since the software formats the array of data in the proper sequence, each piece of information simply falls into place. This is an example of the software simplifying the hardware by removing a data-manipulation burden. A total of 590 bits are transferred in this manner, consisting of 53 words of varying widths.

For transferring the measurements back to the results software, a 10-bit reverse path is provided. Nine transfers are required to obtain the timer and counter data, and another 1024 empty both of the buffers. The 6-bit counter used for loading is also used to select which 10-bit field to transfer. The 20-bit buffers are each allocated two counter "addresses", with the data taken from alternate addresses on consecutive transfers. A large multiplexer is used to funnel all the 10-bit fields into a single reverse path.

SECTION VI

FUTURE TRENDS

Many lessons learned in the design and construction of the HPMD can suggest and guide future work in this area. The primary lesson is the success of the hybrid hardware-software approach. Man and digital circuitry function at such radically different levels that interaction between them is difficult. Program measurement in the hardware differs radically from the programmer's conception of debugging or optimization. Humans, with a limited short-term memory capacity, are severely handicapped if they must recall many minor machine-to-program transformations; those transformations should be performed for them by the debugging and optimization tools.

All detection schemes must deal with the conflict between resolution and complexity. To obtain resolution, many different program parameters must be examinable, with complex operations performed on them to discern specific or obscure relationships. Such power is costly and complex. One of the lessons learned from the HPMD is the size differential between detection based on single signals versus that based on 16-bit parameters. It is necessary to limit the handling of word-size information in detection functions if cost and complexity are important.

Many of the HPMD design and construction difficulties can be traced back to a lack of "monitorability" in the host computer hardware or inflexibility in its systems software. There is a small trend towards providing probe points for assessing the workload on each computer system element, but a device which performs program measurement goes far beyond this. It seems unlikely that computer manufacturers would alter their hardware and software to accommodate external program measurement devices. Long before external devices became a force in computer marketing, internal instrumentation would most likely appear which, if designed by the manufacturer concurrently with the computer itself, could offer the most straightforward and powerful program measurement capability. Internal instrumentation would appear the most effective approach to widespread use of an HPMD-like aid.

With the increasing use of LSI and GSI (large-scale and grand-scale integrated) circuitry, accessibility to the hardware will diminish to nearly zero. An external HPMD will become impossible, but the reduction in hardware cost might very well make practical the inclusion of HPMD capabilities on-chip at the time of manufacture. Such additional on-chip functions should become

increasingly common as the difficulties and expenses of providing them external to the chip become prohibitive.

An alternative to the strict signal-sensing HPMD approach is to implement a computer-cum-monitor in a single firmware package. Some or all of the difficult task of discerning parameters could easily be microprogrammed into the instruction-level architecture. External hardware may or may not be needed, depending on the system resources available for this use. Note that interference, as previously defined, is not inevitable in such an approach; if the firmware relies on external hardware for some detection, reduction, and storage chores, then the firmware's monitoring functions could be "turned on" continuously, without altering the machine behavior when they are used.

The availability of a CPU for detection presents intriguing possibilities. Instead of merely detecting equalities as in the HPMD, far more sophisticated relationships could be detected. These may include ranges, Boolean, or even arithmetic relationships between parameters. If necessary, an entire software subroutine could be executed to determine if the current conditions constitute a program event. Such schemes, if implemented with the firmware or software of the host computer, will cause some speed degradation. Some Boolean or even arithmetic relationships may be detected at full speed in external hardware. Complex sequential algorithms, however, cannot be executed in real time while the monitored program continues to run unless there is a tremendous difference in the relative speeds of the two processes.

The nature of the computers to be monitored are changing, not just in hardware technology or architecture, but in terms of user-visible innovations such as higher-order languages and virtual machines. If an HPMD can be justified because it allows a programmer to solve problems without mentally "switching-gears" to deal with machine-level measurement considerations, then the nature of such measurement devices must evolve with the computer. For debugging a higher-order language, instruction locations should be expressable as labels or line numbers, not object code locations or actual memory addresses. Variable names should be usable for specifying operands, and logical unit numbers for specifying devices or files. In general, the event specification and information reduction must be as symbolic as the language they are supporting, for the programmer deals with the problem symbolically.

Indeed, as operating systems play more and more of a support role for user programs, the debugging and optimization process for programs will merge with that for systems. Software making correct

and efficient use of the system resources is becoming an important part of system optimization; as software becomes more resource-oriented instead of instruction-oriented, program debugging and optimization becomes another aspect of system performance measurement.

Virtual memories vary in their visibility to the user, from named segments to the automatic emulation of near-infinite memories. To the extent that the virtual memory is visible to the user and is involved in a program bug or optimization problem, then the measurement device must not obscure that visibility. If excessive page faults are degrading the performance of a particular routine, then the measurement device must reveal this phenomenon, not gloss over it as an "implementation detail". For virtual memories, cooperation between the processes on many levels of the computer system, from the compiler and relocatable loader to the page fault handler, is needed if such a measurement device is to keep track of a program scattered throughout disks and core pages.

These two examples, higher-order languages and virtual memories, illustrate several general considerations concerning future program measurement devices. First and foremost is the need to keep the man-machine interactions at the human level rather than the machine level. The machine must be visible only where problems related to its nature and functions might arise. The architecture of the measurement device itself should never be visible, although its limitations must be made clear.

Secondly, the evolution of computers is leading us toward a greater interdependence between the levels within an architecture. Virtual memory is a simple concept; its implementation, however, frequently involves a synergy between software, firmware, hardware, and peripherals. As program measurement is included in such an arrangement, it becomes clear that cooperation between it and many other portions of the host computer on many levels will be needed. An effort of this scope should be done during system design rather than after-the-fact.

SECTION VII

CONCLUSIONS AND RECOMMENDATIONS

CONCLUSIONS

The design and construction of the HPMD have shown that the most effective balance between hardware and software complexity is to design very complex hardware while leaving the software design effort at a more manageable level. Obtaining and manipulating hardware word-width parameters was found to be difficult for even modest functions on a minicomputer not designed to be monitored. The software's task was much easier, since data manipulation and interactive command interpretation are common software chores.

Many of the major problems encountered, such as synthesizing parameters and implementing breakpoints, exist only because commercial computers are not currently designed to be monitored. If these problems are attacked early in the design of the computer itself, then they can be eliminated. Most of their causes, such as the inaccessibility of key signals and registers or software inflexibility, are not fundamental. If "monitorability" is considered important, it can be achieved.

The ideal time to consider the connection of program measurement hardware to a computer is at the time of the computer's initial design. The CPU, memory, and I/O buses, as well as the timing and control signals necessary to interprete them, must be made available to external devices. Some decoding is needed for both the execution and the monitoring, such as the direction of information flow, and such information must be explicitly available to the external hardware, rather than buried deep on a board or implicit in a complex signal protocol.

Interrupts must provide instant response to implement a breakpoint. The detection of a breakpoint lags the actual event, and so the halting signal by its very nature is not available until late in an instruction. The resultant interrupt must nonetheless be processed before the next instruction to avoid the "skidding" effect. This is not technically impossible, but is much quicker than currently demanded for handling I/O.

Analogously, the computer's software can ideally be designed with this same foresight, anticipating the demands of the measurement software. The HPMD effort has shown that in many cases features are included within an operating system to protect one

program from the foibles of another. User interrupt definitions are an obvious example; on the NOVA 800, they are wiped out before control is passed from one program to another. Although advantageous in many situations, such protection features make it difficult or impossible for a debugging program to manipulate and control the operating environment of a monitored user program. A little foresight could loosen such constraints which constitute major obstacles to program measurement.

RECOMMENDATIONS

Based on these conclusions, two recommendations can be made to capitalize on the HPMD experiment. A one-year evaluation of instruction-level monitoring techniques, such as provided by the HPMD, is needed to judge the usefulness of the various measurements. The design and construction of the HPMD was a necessary first step to determine what can be done, what measurements are easy to obtain, etc. A perspective on the usefulness of each capability is needed next. This perspective can be the basis for specifying monitoring functions which will truely be assets in software development, optimization, and debugging. This is the ultimate payoff for the HPMD effort.

The final recommendation opens the door for such instruction-level monitoring hardware. If the Air Force actively encourages "monitorability" and even built-in monitors, then such aids may become practical on a wide scale. A computer to which monitoring hardware may be easily attached (or which already includes such hardware) is more suitable for program debugging and optimization. As was learned with the HPMD, it is impractical to expect such powerful tools to be readily added after the fact if their attachment is not anticipated in both the original hardware and software. Active encouragement of this anticipation during design can be a way to insure that such monitors may be later applied as needed.

SECTION VIII

SUMMARY

The HPMD was an experiment in hardware-assisted debugging and optimization. The device itself was designed to make effective program measurements and to be easily controlled by the user. The combining of hardware and software was the key to its success. Their complementary capabilitities provide a design too complex and costly for hardware alone, yet more effective than a purely software tool.

The design goals for effectiveness and complexity appear to have been met in most cases, although an evaluation phase will reveal much more. It would be difficult to devise substantial improvements on the conciseness, clarity, or sheer speed of the software syntax and keyboard characteristics. The invisibility of the hardware architecture was achieved in one sense but not in another. The syntax expresses measurements in quasi-English rather than switches and push-buttons, and the allocation of detection groups is automatic. One the other hand, the measurements themselves correspond exactly to the hardware functions, and some options are specified in a sloppy manner, such as "A,D,#, or I"; they represent the intrusion of hardware quirks into an otherwise elegent command syntax.

The hardware-software approach does prevent any form of interference during measurements. The computer's resources are heavily used to initialize the device and to process the results, but during the actual measurements the hardware does not affect the host computer. The implementation of these general measurement types was found to be not as transferable as originally expected. The concept of a "universal" measurement device interfaced to specific computers proved quite workable, but that interface, at least for the NOVA 800, is unexpectedly large (20% of the device) and is a non-transferable design effort.

Much can be learned from the problems encountered during the design of the HPMD and its resultant shortcomings. Hardware difficulties in obtaining the needed signals, synthesizing the parameters, and implementing breakpoints all stem from the unmonitorable nature of the host computer. Likewise, systems programming difficulties are due to aspects of the operating system which were included as protection for user programs. A major result of the HPMD effort is the concept of "monitorability", and the definition of its hardware and software implications.

Several areas of future growth may be perceived based on the HPMD experiment. Symbolic debugging is a natural approach for a hybrid monitor, since the transformation from symbolic to machine level is a common software function. Use of the host CPU for detection functions is a possibility, as is implementing some or all of the detection functions in firmware.

Two recommendations may be made to direct further growth in this field. Evaluation is needed to determine which program-level measurements are the most productive and worthwhile. Such evaluations permits the proper evolution of monitoring devices based on their potential contribution. Finally, the Air Force should actively encourage the "monitorability" of the computers they specify and purchase. The inclusion of built-in monitoring aids which can interact with controlling software should also be considered. The HPMD has shown the feasibility and advantages of this hybrid approach.

APPENDIX I

SOFTWARE

The following functional descriptions define the basic actions each software routine performs. They are presented here to give the reader a feeling for the scope of this effort and to provide insights into the problems encountered and solutions found.

INITIALIZATION SOFTWARE

DEBUG

This is the main-line for the initialization software; its task is to obtain commands from the user and delegate their execution to the appropriate subprogram, as well as managing the measurement specifications as they accumulate.

As the first code to be executed, this main-line program must OPEN all the files needed for initialization; these files are the (spooled) terminal input and terminal output, and also $HPMD.MS. Upon entering, the most recent measurements and annotation must be recalled, if there are any. By sensing the size of $HPMD.MS, the main-line can determine whether it contains this information or was just created by the OPEN. If empty, the measurement specifications are zeroed and the annotation set to be a null string. Otherwise, the old measurements and annotation are read in from disk, and the annotation is printed at the terminal as a reminder to the user of the nature of those measurements.

Next, a command is obtained. The user is cued with a ":", the current measurements preserved in a temporary array, and NEWCHAR is called to obtain the single character which begins the command. The colon cue is unique to this initial point in the syntax. NEWCHAR, which manages the keyboard, is instructed to accept only a character in the string "CTSGHDKURAQP", the first letter of each command. Based on this first character, the appropriate subprogram can be called to perform the specified function. Before this happens, DEBUG checks to see if the command is RUN or QUIT; if it is one of these, no more measurements will be added and so the older information in $HPMD.MS is deleted and replaced by the current measurements. This insures that $HPMD.MS contains only the latest information between DEBUG executions. The first character is then checked to see if it is "D", "P", "A", "U", or "R". These represent commands which do not add measurements, and therefore can always be

performed; DEL, MPRINT, ANNOT, KEEP, USE, or RUN is called respectively. Otherwise, a measurement is to be added. COMOK is called to determine if there is room within the HPMD hardware to also handle the new measurement; if there is not, DEBUG prints a message for the user. Otherwise, the "C", "S", "G", "T", or "H" will cause the execution of COUNT, STORE, GRAB, TIME, or HALT respectively.

If at some time during the measurement specification dialogue the user types a line feed, then the function is aborted and the cancel flag is turned on. Before branching back to the beginning to obtain a new first character, this flag is checked. If it is ON, then a "...NOT DONE" is printed to assure the user that the software handled the line feed properly, and the measurements saved in the temporary array are recalled; if it is OFF, then the new measurements are preserved and, assuming QUIT was not specified, the next command is processed.

COUNT

This is the simplest of the measurement-specifying routines, since the syntax for a Count command consists only of the "Count if" and an event definition. Upon entering one of these routines, the first character has already been echoed to the terminal, and so COUNT prints only "OUNT IF". NEWVALS, which reads in event definitions, is then called with parameters which indicate doubling for this command is not allowed, and that the event obtained is to drive a counter. Since NEWVALS both obtains and saves the parameter and values which define the event, COUNT has nothing else to do. It should be noted that NEWVALS returns a cancel flag which indicates whether the user aborted the event definition with a line feed. Count, like all other commands, returns immediately if the flag is on, passing it back to DEBUG.

GRAB

The syntax for the Grab command differs from the other commands in that the type of information to be saved in the buffer is specified by a "A", "D", "#", or "I", and three questions about the Grab buffer control are asked. First of all, "rab current" is printed since the "G" has already been echoed. NEWCHAR is then called, with a "AD#I" string as its legal characters. The character obtained is decoded into 1, 2, 3, or 4 respectively and saved as part of the current measurements. Assuming the cancel flag was not set by NEWCHAR, " IF " is printed and NEWVALS is called. NEWVALS is instructed not to allow doubling and to save the event obtained as a Grab event. To handle the questions and answers for controlling

both buffers, QA is used. QA is passed a "question" string and where to put the TRUE/FALSE answer. GRAB therefore calls QA three times with the questions: "FREEZE BUFFER WHEN FULL?", "INHIBIT IF STORE BUFFER STOPPED?", and "HALT PROGRAM WHEN BUFFER FULL?". As with any routine which obtains information from the user, the cancel flag may be set by QA.

TIME

The time syntax includes two events, either or both of which may include doubling. "IME FROM" is printed, and then NEWVALS is called to get the first event. COMOK informed DEBUG, upon "T" being entered, that a time measurement was within the capacity of the HPMD hardware. This did not imply that doubling is possible. To perform the analysis to determine if there is capacity for a doubling, DOUBLE can be called. The flag returned is passed to NEWVALS, which obtains and saves the first event, whether doubled or not "UNTIL" is then printed, followed by DOUBLE and NEWVALS to handle the second event.

STORE

STORE is similar to TIME, except that the parameter to be stored must be entered and there are buffer control options, as in GRAB. "TORE" is printed to complete the command name, and PARAM is used to get the parameter. PARAM is a lower-level routine used also by NEWVALS, which obtains the specified parameter from the terminal and returns a number 0-15. In this case, PARAM was instructed not to accept a "B" (for "Both"), since this parameter entry is quite distinct from the doubling event syntax case which NEWVALS must handle. Assuming the cancel flag is not turned on, "FROM" is printed and then DOUBLE and NEWVALS are called. As with TIME, DOUBLE determines if doubling is legal for a Store command given the current measurements, and NEWVALS gets and saves one or two events accordingly. "UNTIL" delimits the start event(s) from the stop event(s), and the DOUBLE and NEWVALS procedure is repeated a second time. All of these calls to NEWVALS are dependent on the cancel flag remaining off. QA is then used to handle the four questions about the buffer control options. "FREEZE BUFFER WHEN FULL?", "PROHIBIT RESTARTING?", "HALT PROGRAM WHEN BUFFER FULL?", and "INITIALLY RUNNING?", are asked and the user's response is saved as part of the current measurements, again assuming no line feeds.

HALT

The last of these five routines which specify measurements is HALT. This combines the single event of COUNT and GRAB with the

possibility of doubling as seen in TIME and STORE. "alt program if" is printed, and then DOUBLE and NEWVALS are used to get and save the single or doubled event.

QA

QA poses a question and decodes the user's answer. It is called by STORE and GRAB to interrogate the user about the buffer control options. The flag bit corresponding to the specified buffer option is first turned off by ANDing the flags with a mask, and the question string passed to QA is printed. NEWCHAR is then called, with "NFYT1O" given as the string of legal characters. These allow the user to think of the TRUE-FALSE answer as "TRUE" or "FALSE", "YES" or "NO", "1" or "0", even "YEP" or "NOPE". Assuming the cancel flag is not set, the returned character is checked to see if it is "Y", "T", or "1"; if it is, then the flag bit is turned on, if not, the flag remains OFF.

COMOK

COMOK combines arithmetic and Boolean operations to determine if a particular command is possible given the current measurements. The algorithm must deal with the difference between the measurements as specified and the measurements as assigned detector groups upon a Run command. For Time, Store, and Grab, there is no difference; only one detector group is capable of doing these measurements' specialized tasks. This is not true, however, for Count and Halt. There are three identical counters, and any leftover detection groups can control the halt interrupt. It would be a shame to assign specific detection groups for these measurements any earlier than necessary, since such an assignment may preclude a subsequent measurement. For example, if a Halt is specified first and is assigned a particular group, say Group #1, then the timer could not be used. Likewise, if a Count was assigned to Group #5, then the "Start timer" event could not be doubled (that would need #1 and #5).

To avoid this lack of versatility, the measurements are stored as they are specified, without any assignment to particular detection groups. COMOK is called to determine if there is room for any new measurement being entered by the user. If there is not, then the DEBUG main-line will print "INSUFFICIENT CAPACITY FOR THIS MEASUREMENT....DELETE A MEASUREMENT" to tell the user that there is not enough capability in the hardware to accomplish both the current measurements and this new one.

The algorithm COMOK uses to make this determination analyzes what room is available and what the new measurement demands. First the initial command character, "T", "S", "C", "G", or "H", is identified and converted into an integer. The number of non-Halt measurements is counted, and the number of Halts and double-Halt groups needed is subtracted from it; this gives the space for additional Halts. If the measurement requested is a Halt and this space number is greater than zero, then COMOK returns as True. Otherwise it must differentiate between Time and Store, which use two groups each, even without any doubling, and the Count and Grab, which need only one. If there is room for at least two more halts, and neither time group is currently needed, then the time measurement is legal; the Store measurement is handled analogously. If there is room for at least one more halt, and there are less than three Counts specified currently, then another Count is legal. Also, assuming no Grab measurements are specified, space for a halt implies space for a Grab. In this way, COMOK determines if a measurement is possible.

COMOK returning TRUE implies only that the command can begin to be entered; if doubling is demanded later in the command, then the user may yet be prohibited from getting the command as desired. DOUBLE performs a similar function for TIME, STORE, and HALT by checking the legality of doubling for that command given the current measurements.

DOUBLE

DOUBLE must also deal with the specification/assignment dichotomy. Only Times, Stores, and Halts can be doubled, and DOUBLE determines if this is possible for a particular command and the current measurements. The algorithm used first calculates how much room there is left for halts, which can go anywhere, and then the room for double-halts. If there is not room for at least two single halts and the possible unused groups arranged so at least one double-halt can be added, then DOUBLE returns FALSE. Next, further checks are made for the TIME and STORE commands: if doubling is requested for either time event or the start STORE buffer event, then DOUBLE checks to see if there are less than three counters and previous doubles for these events. If so, then what counters there are can be shifted into assignments which have room for this additional doubling. If a stop STORE buffer doubling is requested, then it is permitted unless there is a Grab measurement specified.

DOUBLE performs a function similar to COMOK, but it is used differently. COMOK is used to check if a command already begun should be ruled out; DOUBLE is called before the user can type "B"

for "Both", and therefore, determines whether that key should be legal or not when the event is defined.

NEWVALS

NEWVALS handles the definition of all events. PARAM and GETVAL can obtain the parameter and each octal value; NEWVALS main task is therefore to manage their use and to store the results in the proper format. It is passed the type of command and which event within that command it must obtain, as well as whether doubling is permitted. Based on these it gets one or two parameter-values definitions and adds them to the current measurements.

PARAM is called first, followed by printing "=". The parameter is stored in the measurements, as is an indicator defining the type of event. GETVAL is called up to four times or until it sees a carriage return, and the values obtained are stored. If doubling was prohibited or PARAM did not receive a "Both" before the parameter, then NEWVALS returns. Otherwise, the parameter-values process is repeated once more, with doubling definitely prohibited and the information obtained is stored as the "doubling" of the earlier event type.

PARAM

Although PARAM is the third largest routine in the initialization software, its task is quite simple. It must obtain from the user a parameter mnemonic as defined in Table I and returns the corresponding sequence number, 0-15. If doubling is allowed for the event which this parameter is a part, then "Both" is allowed and PARAM returns an indicator as to whether it indeed preceded the parameter. Finally, as will all routines which interact with the user, the cancel flag is returned to indicate whether a line feed was typed.

To obtain the parameters yet maintain the feature of requiring only the minimum of typing, a tree structure is searched, based on the first, second, and third characters. As soon as the characters inputted uniquely define a parameter, PARAM prints the remaining character(s) and returns with the sequence number. For example, if "A" is typed, only one parameter ("AST") is possible, and so PARAM prints "ST" and returns with an 11. NEWCHAR is used to full advantage by PARAM since at each node in the tree structure there are only a limited number of legal next characters, and NEWCHAR can control the keyboard to limit the user to that set. The tree thus consists of calls to NEWCHAR with the legal next characters for

every non-ending node, and a sequence number and possible printing string at each ending node.

GETVAL

GETVAL returns a value derived from the octal number typed by the user. To do this it must both perform ASCII octal digit to integer conversion and detect the end of the number; the latter task is more complex. In addition, a rubout is interpreted as a "local cancel", allowing the user to start over in defining the value, and line feed again cancels the entire command.

As an event is defined, NEWVALS calls GETVAL up to four times in a row to get a group of values until a carriage return is entered. A value must be returned the first time; subsequent values are optional. A flag is passed to GETVAL indicating whether to accept a carriage return instead of a value. GETVAL in turn returns a description of what happened: carriage return alone (if legal), value without carriage return, and value with carriage return. A value is delimited by ending with a slash or a carriage return; if a value exceeds octal 20000, further digits cannot be added and the keyboard must be limited to "/", carriage return and rubout. This avoids any "value too large" error message.

NEWCHAR is used to obtain the input characters. At first, depending on whether a value must be entered, "0-7" or "0-7" and carriage return are provided as the legal characters. After the first character, "/" is added to the list, as is carriage return if it is not already there. The characters are converted to numbers and added to a running integer total; the integer is multiplied by 8 before adding in each new number. Upon a slash or carriage return, the integer is the octal value to be returned. Upon a rubout, the integer is zeroed and the process begun again.

DEL

DEL handles the Delete command, one of the simplest to use, yet quite difficult to implement. The other routines store the measurements in a long array, in the order they were entered. This array must be edited to remove any measurements the user wishes to delete.

First, DEL determines which types of measurements are currently specified, and constructs a string of characters each of which begins one of the corresponding commands. Thus if there is a Time, two Counts and a Grab the string would be "TCG". "A" is added to the string, and it is then used in a call to NEWCHAR as the list of

legal characters after "elete" is printed. NEWCHAR returns one of these, and the measurement editing is based on the character. The remainder of that command is immediately printed, such as "ime", "ount" or "rab".

If "A" was returned, all of the measurements are erased. If "T", "S", or "G" was returned, the array is searched for the (only) entry which corresponds, and it is eliminated. If it was listed as the first half of a doubling, the other part is found and erased also. Counts and Halts can be different, since there can be more than one of each. DEL checks how many Counts or Halts are specified, and if there is only one, it is found and erased. For Halts, any doubling is also found and erased. If there are more than one, a "#" is printed to prompt the user to enter a number to indicate which measurement is to be deleted. NEWCHAR is called, given "123" as legal digits for a Count and "12345678" for Halt. The array is then scanned and when the corresponding Count or Halt measurement is found, it is erased.

The erasing operation is performed by an internally defined procedure, which overwrites the measurement being deleted with the remainder of the array, shrinking the array in the process. The number of measurements is decremented, as is the count of that particular kind of measurement.

ANNOT

ANNOT builds a long character string which the DEBUG main-line saves in $HPMD.MS for printing by the results software or upon reentering DEBUG. KEEP and USE also save and recall the string along with the other information in their files. "nnotate:" is printed first, and then the user can enter the comments using any alphanumeric and most punctuation keys. Two carriage returns in a row terminate the comment. Line feed cancels the entire annotation, preserving the earlier one, if any, and rubout deletes the previous character from the string.

The string has a maximum length of 5000 characters. If the user attempts to enter further characters, NEWCHAR limits them to carriage return and rubout. A "?" provides the explanation: "No more room; valid characters are CR and rubout".

MPRINT

MPRINT adapts MLIST, which generates a formatted description of the current measurements, for listing in the terminal or line printer. First it checks to see if there indeed are any

measurements to print; if not, it responds "rint...no measurements currently specified". If there are some, it prompts the user with "rint measurements at" and calls NEWCHAR to obtain a "T" or "L" for Terminal or Lineprinter. Assuming the cancel flag is not entered, it either writes "erminal" or "ineprinter", and if the latter, the line printer is OPENed as an output device. MLIST is then called, given the appropriate channel number, and the line printer is closed if this is needed.

MLIST

To generate a readable printout of the current measurements, it is necessary to reverse the encoding of the command by the DEBUG main-line of the parameter by PARAM, of the values by GETVAL, and of the buffer options by QA. MLIST does this by manipulating character strings based on the encoded measurements. The results are written on an output device or file as specified by the calling routine. The reconstructed printout looks quite similar to the original terminal session when the measurements were first entered interactively.

The current measurements are stored as array entries consisting of type, parameter, and values. These entries are outputted in order, with each type getting a different format. First, the command itself is printed, such as "Time from" or "Halt if". If it is a Store or Grab, the parameter or "AD#I" selected is decoded into the proper character(s) and outputted, followed by a "from" or "if". If the type of entry is the second half of a Time or Store, "until" is substituted, and if it is the doubling of the previous entry, "...and" is used. Next, the parameter is decoded into its Table I mnemonic and outputted with an "=". Finally OCTPRINT is called four times to output the values in octal, separated by "/". If the command was a Store or Grab, concise definitions of the buffer options are added, such as "Freeze", "Halt", and "Once". The results of MLIST for a sample set of measurements are shown in Figure 3.

OCTPRINT

OCTPRINT outputs to the specified output channel six digits which represent the octal equivalent of the value passed to it. Sign-testing and mask-and-shift are used to generate six integers which equal the value of each octal digit field in the passed value. Writing them without intervening blanks produces the octal number.

```
TIME        FROM  IA  = 002056 / 002056 / 002056 / 002056
            UNTIL IA  = 002073 / 002077 / 002077 / 002077
COUNT       IF    IA  = 002056 / 002056 / 002056 / 002056
COUNT       IF    OU# = 000033 / 000033 / 000033 / 000033
STORE INT   FROM  IA  = 001753 / 001753 / 001753 / 001753   RUN
            UNTIL IA  = 003007 / 003007 / 003007 / 003007
```

Figure 3. Sample MLIST Output

RUN

RUN is the largest single routine in the entire HPMD software. It has four responsibilities: assigning actual detection groups for the current measurements while converting them into HPMD hardware format, determining a set of flags for the results software which indicate which parts of the hardware are being used, loading these final measurements into $HPMD.MS, and finally creating a suitable execution environment for the user's program. SETUP is then called to actually run the user program.

Converting the measurements from DEBUG measurements array format to HPMD hardware format occupies the first half of RUN. As seen in COMOK and DOUBLE, the measurements are not assigned specific groups as they are entered. Only after the user types "R" can this assignment be done, since no more entries will be made. Time, Store, and Grab measurements must go in specific places; they are not flexible in their assignments. They are therefore assigned their detection groups first. Double Halts are somewhat flexible, and are assigned next. Any pair (1-5, 2-6, 3-7, 4-8) of unused doubled detection groups can implement a double-Halt, but it is not until the Time, Store, and/or Grab assignments are known that a leftover pair can be located. Counts come next: whatever groups among #5-#7 that the Time, Store, and double-Halt assignments did not use are available for Counts. Finally, the single Halts, which can go anywhere, are assigned whatever group(s) remain. The sequence of fixed measurements, double-Halts, Counts, and single-Halts insures a conflict-free assignment. COMOK and DOUBLE have already checked to make sure the measurement entries can be squeezed into the hardware without exceeding its resources. This process of assignment generates another array, called TABLE, which contains the information to be loaded directly into the HPMD hardware. The

hardware is not informed the meaning of each word of information; its position in TABLE implies its meaning. For example, positions 33-40 are the detection parameters for groups #1-#8 respectively, while positions 48-50 contain the buffer control options. Such a rigid format makes the hardware's task easier.

The flags for the results software are set while assigning the measurements, since at that time the use of specific hardware groups is known. These flags will allow the results software to, for example, omit printing the count in counter #1 if its detection group is being used to double the start Timer event instead. This is the only point within the entire initialization software that such a determination is known.

$HPMD.MS is now written with all the information available about the current measurements. TABLE, the measurements array, and the flags for the results software are put in first. COM.CM, the file in which the operating system puts "DEBUG" and the program name to be executed is read to make sure that a program was indeed specified; the program file is OPENed and CLOSEd to insure that it exists. Error messages tell the user if either of these tests fail. The program name is then written into $HPMD.MS; all old entries except the current annotation have thus been overwritten. Skipping over that annotation, the file pointer is advanced to the end-of-file, and MLIST is called to add a printable listing of the current measurements for the results software. Because the annotation and measurements are the last things in the file, the results software can print them together in a simple read-print loop which can continue until the new end-of-file is reached.

COM.CM, from which the user program name was extracted, is edited to remove the word "DEBUG". When this is done, it is identical to what it would have been if the user had typed simply the program name with any parameters and switches. The environment for the user program is therefore as the program expects them in terms of all those parameters and switches, and the user program will behave normally.

RUN has now done its tasks and can call SETUP. SETUP does not return to RUN, and so RUN does not return to the DEBUG main-line, except if errors were encountered with the user's program name.

SETUP

SETUP is an assembly language routine which simply chains to the RERUN program via a system call. Originally intended to do all the outputting to the HPMD hardware itself, SETUP was reduced to a

system call when RERUN began to overlap such duties. RERUN will read TABLE out of $HPMD.MS and load it into the HPMD hardware. It will then swap in and execute the user program, and finally run the results software.

KEEP

To save a copy of the current measurements in a disk file, KEEP calls GETNAME to obtain the file name. After appending ".MS" to indicate it's a measurement file, it is OPENed to see if it already exists. If it does, it is deleted since it might be longer than the current information. Then it is recreated, and the current measurements array and annotation are written into it.

USE

USE performs a function complementary to that of KEEP. GETNAME is used to obtain the file name, and ".MS" is again appended. If successfully OPENed, the measurements and annotation are read to replace the current ones. If the OPEN is unsuccessful, "...MEASUREMENT FILE NOT FOUND" is printed and the current measurements and annotation remain unchanged.

GETNAME

To obtain a file name, GETNAME uses NEWCHAR to read in up to 11 alphanumeric characters. A "$" is not allowed in such a file name, thwarting the user who attempts to apply KEEP or USE to $HPMD.MS, which has the wrong format. Assuming the cancel flag is not set, they are checked for a rubout or carriage return. As in ANNOT, a rubout is echoed as a backarrow and the previous character is deleted; if a carriage return is seen, GETNAME returns with the file name. When 11 characters have been entered without a carriage return, NEWCHAR is limited to only rubout and carriage return, with an explanatory message available upon a "?".

NEWCHAR

Last but not least is the routine which handles all terminal inputs from the user. Called by DEBUG, QA, PARAM, GRAB, DEL, ANNOT, MPRINT, and GETNAME, it manages the keyboard and thereby implements the legal-illegal keys features, the line feed cancel feature, as well as "?" and "!" responses.

To do this, it simply reads in a character, checks it against the legal character string passed to it plus "?", "!", and line feed and, assuming it matches one of them, it acts accordingly. If no

match is found, it loops back and reads another character without echoing the first one; in this way, the former key appears dead to the user. If it is a "?", the explanation message specified by the calling routine is printed; if it is an "!", then the legal string passed is printed instead. In either case, NEWCHAR then loops back to read another character. If it is a line feed, the cancel flag is set. Finally, if it is a legal character, then that character is echoed, and thus it appears to the user to function normally. Upon either a legal character or a line feed, NEWCHAR returns.

EXECUTION SOFTWARE

RERUN

To set up the execution and measurements, RERUN first loads the HPMD hardware. $HPMD.MS is OPENed, and the first 130 bytes are read into core. These consist of 114 bytes comprising the hardware-loadable TABLE, plus the name of the user program, and the flags passed primarily to the results software. The TABLE array is then outputted to the hardware, after the hardware has been cleared by an NIOC instruction on the NOVA 800. One of the flags passed from the initialization software reflects the Store buffer option of initially started vs. initially stopped. If started is selected, a DIB instruction is executed, enabling the Store buffer; conversely, a DIC instruction will ensure that the buffer is disabled if that is selected. Next, the flag which indicates whether any Halts were specified is checked, because this means HPMD interrupts may have to be handled. If so, the MOVEISR subroutine is called.

An NIOS instruction is executed to start up the HPMD by zeroing the timer and counters as well as enabling any interrupts, and then the user program is executed by a system call (see Figure 4). Since it is a swap rather than a chain, when the user program completes, RERUN picks up from after that system call. Depending on whether the user program makes a normal or abnormal return, the error flag in an accumulator may be saved. Finally, RESTORE and also DISPLAY are executed, and then RERUN returns to the system. If the user's program returned with an error code, RERUN returns to the system abnormally with that same error code.

MOVEISR

MOVEISR consists of two programs: the ISR itself and a program which copies the ISR into core next to the user program. MOVEISR calls subroutine GETSTART, which calculates a starting address for

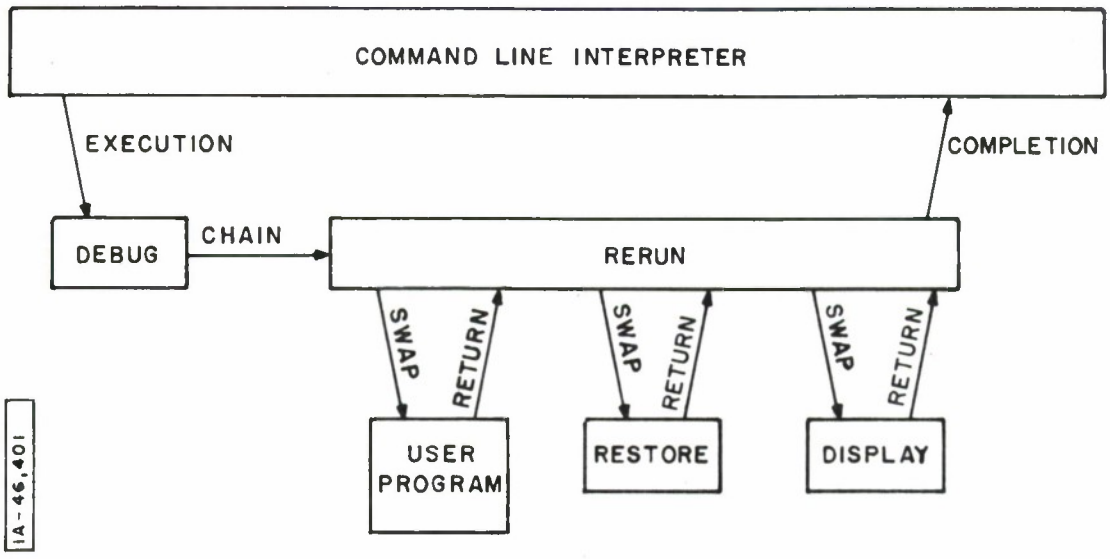

Figure 4 HPMD SOFTWARE PROGRAMS

the ISR based on the size of the user program. To copy it into a new location in core, three byte pointers must be relocated. The byte pointers in the ISR point to file names of the programs to execute: DISPLAY, QZ, and the user's program. They are used in the program swap system calls. The user program's name must itself be copied from its storage in RERUN into the ISR. The system interrupt vector, core location 1, is saved for later use. The actual copying of the "relocated" ISR is now performed, and its starting address stored into location 1. When this is done, all interrupts cause the execution of this ISR.

The relocated ISR checks the interrupt to see if it is from the HPMD; if not, it branches to the system interrupt routine, whose address was saved from location 1. Otherwise, it clears the HPMD with an NIOC 5 instruction, which both disables further HPMD interrupts and zeroes its address register. The status can then be read in, which includes the state of the NOVA interrupt enable/disable flag. In order to allow Halts despite disabled system interrupts, the hardware simultaneously requests an interrupt and forces an interrupt enable. The rest of the machine state, such as the PC and registers, are also saved in core for later restoration.

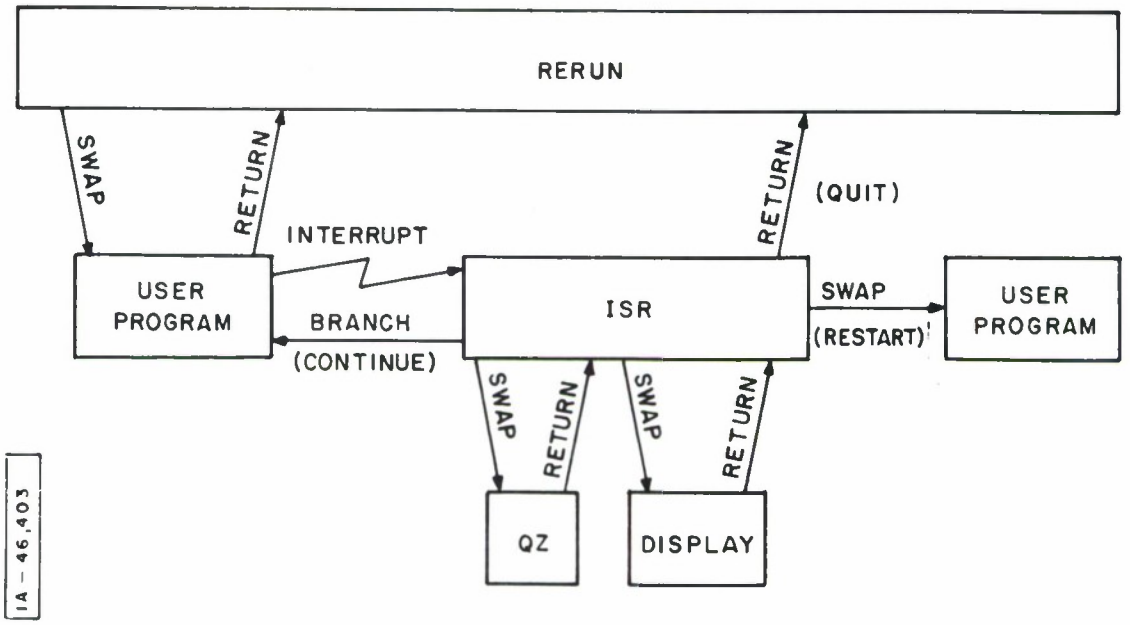

Figure 5 INTERRUPT SOFTWARE INTERACTIONS

QZ is then executed via a system call to query the user about what to do next. Based on whether it returns normally or abnormally, and if abnormally with what "error" value, ISR can take one of several paths (see Figure 5). Upon a normal return, the ISR assumes the user wished to stop any further execution, and so ISR itself returns. Since the ISR execution is invisible to the operating system, this return is handled identically to an actual user program return, and RERUN resumes by running the results software. Otherwise, DISPLAY (the results software) is run by the ISR itself. Again, this is invisible to the operating system, and so the normal restrictions as to what can be done in a conventional interrupt routine do not apply. After DISPLAY completes and returns, ISR either restores all the registers and interrupt enable/disable state and branches to the old PC address, or it chains to the user program. These two actions effect a continuation onward from the halt breakpoint, or alternatively bring in a fresh copy of the user program object code and thus restart the program.

By relying heavily on separate programs both to print the results accumulated and to interrogate the user as to how to proceed, the ISR is kept quite small (81 words). This is

significant since it must rest in high core co-resident with the user program.

GETSTART

GETSTART computes the location of the ISR based on the size of the user program. At a minimum, it must reside beyond location 13000 (octal). If the user cancelled the program under test, the command line interpreter is run automatically to handle user requests from the console, obliterating any code below 13000 long before the user could have it run RESTORE to relieve the ISR of its responsibilities. Similarly it must be beyond the end of the user program. GETSTART reads the copy of the user program object code stored on disk to find the control information defining the highest core address used, and returns to MOVEISR with either one greater than that address or 13000, whichever is higher.

QZ

When the ISR determines that it has received an HPMD interrupt, it runs QZ to ask the user what to do next. Although it is a separate program, it is used as a subroutine for the ISR, which allows much additional code without enlarging the ISR.

As a separate program, QZ first OPENs the terminal input and terminal output, and then prints "Enter ´C´ to continue, ´R´ to restart, or ´Q´ to quit". A single input character is read in, and compared with a set of six possibilities, "C", "R", "Q", "?", "!", and line feed. If no match is found, the character is ignored and QZ branches back to read in another. This imitates the user environment created by NEWCHAR in the initialization software by making unreasonable keys appear "dead". Similarly, "?" and "!" are allowed, but both simply cause the repetition of the initial cue about "C", "R", and "Q". In keeping with the protocol expected by the ISR, "Q" causes a normal return from QZ, whereas "C" and "R" cause abnormal returns with differing error codes. Finally, the line feed is treated as a "Q", aborting the user program execution.

RESTORE

RESTORE is a simple program which stores one of several values in core location 1. A system call which returns the location of the operating system is used to obtain a number unique for each revision of the NOVA RDOS system. By checking it against its possible values, RESTORE can determine which revision is being used, and, therefore, where the normal system interrupt routine is. By storing

the system interrupt routine's normal entrance point into location 1, all interrupts will be handled by it bypassing any ISR.

If the returned location matches no known value, a message is printed saying "SYSTEM CLOBBERED...DO PROGRAM LOAD"; otherwise, "*FINISHED*" is printed to indicate both the end of the user program if RERUN calls RESTORE, or to reassure the user that the interrupt vector was now normal if RESTORE is run manually. It should be noted that RESTORE puts the proper value in location 1 whether or not it is there already. This means that a user can run RESTORE whenever desired for reassurance without danger of doing anything disruptive.

RESULTS SOFTWARE

DISPLAY

DISPLAY is the results software main-line, which does all the read operations from the hardware and calls the appropriate conversion, formatting, and outputting routines (see Figure 6). Immediately upon entrance, an NIOC 5 instruction is executed which disables any possible HPMD interrupt, as well as clearing the address register. DIA instructions can then obtain the information shown in Table V.

The hardware/software interface for reading information into DISPLAY is similar in many ways to that for loading the device. The same internal address register is used to select which information is available. Table V lists the information obtained for each value in the address register. 0-10 can be read repeatedly, since the status, timer, and counters are not changed by being read; the buffers lose the information after it is shifted out of them. This is not a disadvantage, however, since they can be printed as they are read. Indeed, the zeroes simultaneously shifted into the buffers which replace the data are a convenient and automatic way of clearing the buffers before they are used again.

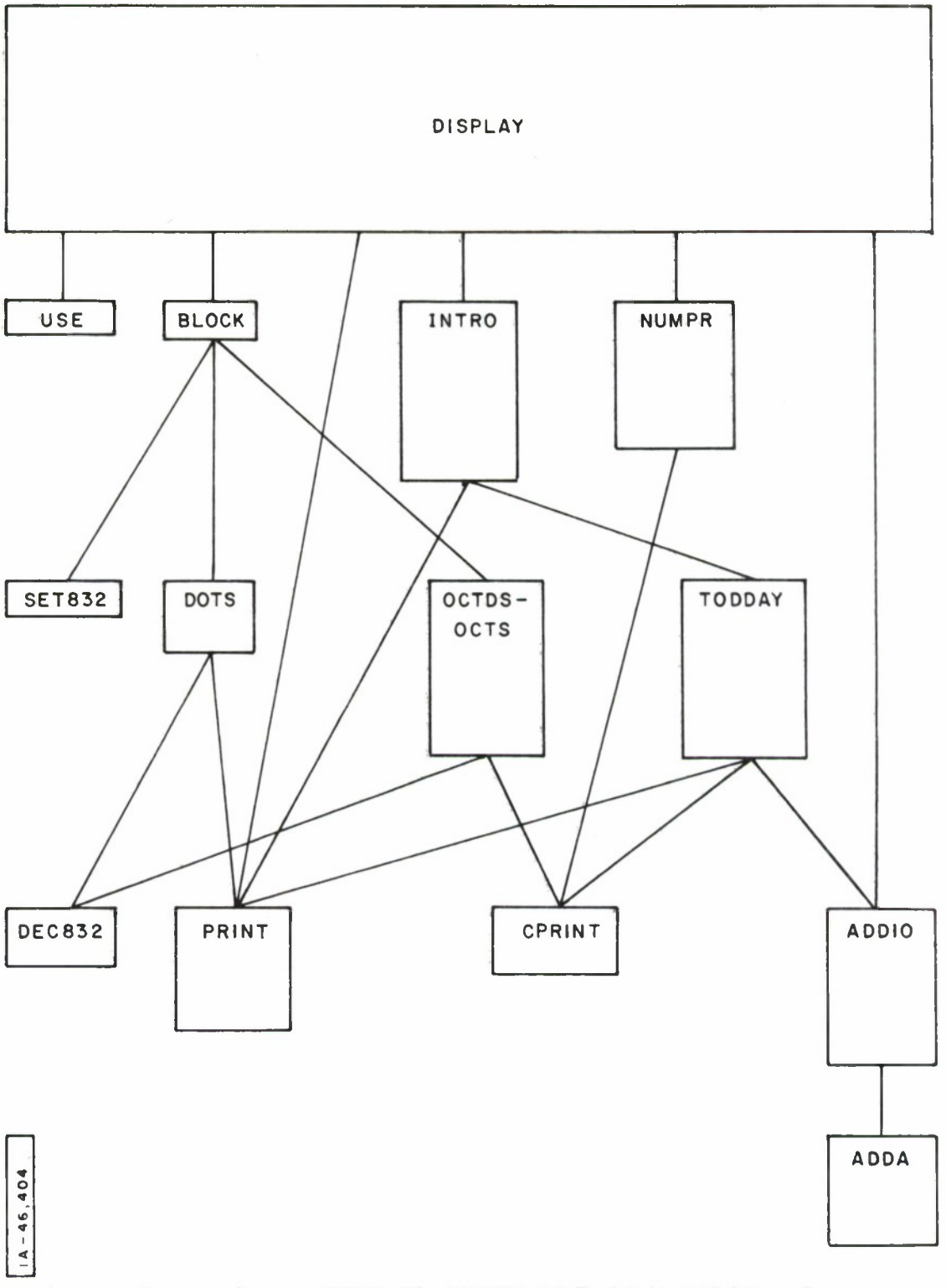

Figure 6 RESULTS SOFTWARE ARCHITECTURE

Table V

Reading Addresses

Address Register	Information Obtained
0	Cable Interlock and Proper ION Flag
1	Timer Bits 0-9
2	Timer Bits 10-19
3	Timer Bits 20-29
4	Counter #1, Bits 0-9
5	Counter #1, Bits 10-19
6	Counter #2, Bits 0-9
7	Counter #2, Bits 10-19
8	Counter #3, Bits 0-9
9	Counter #3, Bits 10-19
10	(spare)
11	(spare)
12	Store Buffer Bits 0-5 + 4 extra bits
13	Store Buffer Bits 6-15
14	Grab Buffer Bits 0-5 + 4 extra bits
15	Grab Buffer Bits 6-15

The status is first read in, but with this current software version, it is ignored. INTRO is then called, which handles all of the heading, time, date, and $HPMD.MS information. INTRO returns several flags which indicate which of the six measurements were indeed specified.

The time measurement is then read in and converted to decimal, 10 bits at a time. The counts and time as obtained from the hardware are simply large binary numbers, 20 and 30 bits respectively. The timer number is a count of 200 nsec. intervals. A DIA gets the highest 10 bits of the 30 bit timer value, and ADD10 converts it into an array of ten decimal digits. A "2" is passed to ADD10, which indicates that the decimal digits must represent twice the value of the 10 bit word. This takes into account the fact that each timer count represents 200 nsec., and so the decimal digits will be the time in tenths of microseconds. The process is repeated twice more, with the new 10 bit values read in and added to the decimal digits as the decimal digits are shifted over by 1024. The result is a digit array which represents, in decimal, (value #1 x 2097152) + (value #2 x 2048) + (value #3 x 2). The 30 bit value has thus been doubled and converted to decimal.

USE is then called to check the flags returned by INTRO. If the timer was specified, USE returns to the next statement after the call; otherwise, it skips to the address given as a parameter, in this case the counter processing code. Assuming the timer value is relevent, PRINT is called to print the string "TIME=" on the line printer. Next, NUMPR prints the decimal digit array in the format specified. A zero is passed to NUMPR, informing it that all 10 digits are to be printed. When all the digits are to be printed, NUMPR uses the format XXX.XXXXXXX, which fits the time measurement. PRINT is called again after the digits to add "SECONDS" and a carriage return.

The count values are handled in a similar way; DIA gets the 10 bit values and ADD10 assembles them into a decimal number. A "1" is passed to ADD10 the first time, since the 20-bit binary value is a straightforward count and does not need to be doubled as did the time. Since there are 20 bits instead of 30, DIA and the ADD10 routine are used only twice for each count. Again, USE is called to skip the printing of any meaningless count. Otherwise, PRINT provides a "COUNT=" and NUMPR prints the decimal digits. Passed a "3", NUMPR skips the first three digits and decimal point, producing a XXXXXXX format. This code for processing the counts is looped through three times, once for each counter.

In the case of both the timer and the counters, the DIA instructions as well as the calls to ADD10 are executed before USE checks if the measurement was specified. This wastes CPU time, but because ADD10 is fast and DISPLAY has time to burn, it is not a significant drawback. By doing it in this manner, the address register in the HPMD hardware is advanced over meaningless times or counter values just as if they were relevant; no extra code is needed to advance the register to replace the normal processing steps.

A DIB instruction forces 10 into the address register, in preparation for reading in the Store buffer's data. USE is then called, since the buffer need not be read in if it is irrelevant. Next, PRINT provides a heading "STORE BUFFER:" for the data, and BLOCK is called to get and print it. BLOCK will read in all 256 values, and print them in an 8 column by 32 row array of octal numbers. "....." will be substituted for the locations in the beginning of the buffer which precede the actual data, if there are any. This process is repeated for the GRAB buffer with no differences except the heading "GRAB BUFFER:". The DISPLAY main-line is now complete, and returns to the calling routine (RERUN, the interrupt servicing routine, or the operating system's command line interpreter).

INTRO

INTRO handles the opening of files and the listing of all the heading information. In a sense, it should be part of the main-line code, but it was separated into its own routine to make the main-line itself less cluttered and more readable. Its chores are few and straightforward, but in assembly language they require over a page.

INTRO first opens the line printer and calls PRINT to create the heading box at the start of the listing (see Figure 1). The line printer will remain open until DISPLAY completes. TODDAY is then called, which adds the time of day and the date to the listing. $HPMD.MS is opened, positioned to the flags which indicate which measurements were specified and these flags are read into core. $HPMD.MS is then positioned to the annotation and measurements, which are stored in the file completely formatted and ready to print. Alternating line-reads from $HPMD.MS and line-writes to the line printer dump them on to the listing until an EOF halts the process. The flags are then loaded into a register for the main-line to use, and INTRO returns.

NUMPR

NUMPR prints an array of digits on the line printer in one of several formats. From one to ten digits can be specified; if ten, a decimal point is inserted between the third and fourth most significant ones. Leading zeros are omitted, thereby left justifying the number. The array is processed digit by digit, the first non-zero one turning off the loading zero suppression flag and enabling their printing using CPRINT, the character printing routine. The digits are converted to ASCII by adding the character "0" to the digit. Exceptions to the zero suppression rule are that the 10 digit format's decimal point also turns off the suppression flag, and it is always turned off before the last digit. This permits a "0" count to be printed rather than a blank one. When all the digits have been processed, NUMPR returns.

ADD10

ADD10 adds a 10 bit binary value to a 10 digit decimal array, putting the result in the decimal array. A parameter passed to ADD10 defines the method in which this is to be done. If the parameter is 1, then the array is zeroed before the addition, and therefore, the binary number is converted to decimal. If the parameter is 2, the array is zeroed as before, but the binary number is doubled before the addition; the result is an array decimal

digits equal to twice the binary value. If the parameter in a subsequent call is 0, then the current decimal array is multiplied by 1024 before having the binary value added to it. This allows consecutive calls to accumulate the decimal equivalent of a 20 or 30 bit binary value by successive decimal shifts and additions.

The key to the algorithm which performs this binary-to-decimal conversion is ADDA, a routine which adds two arrays of decimal digits. By providing it with a decimal array containing 1, it can add that array to itself repeatedly, thereby generating the powers of two in the form of decimal digits. This power array can be added to the other decimal digit array selectively, depending on whether the binary bit in the 10-bit binary word which corresponds to that particular power of two is TRUE or not. For example, if the binary number was 49, the power array would be added in only when it equalled 1, 16, and 32; the lowest decimal digit would have a 1, 6, and 2 added to it, with the next digit getting 0, 1, and 3. In this way, binary to decimal conversion can be accomplished easily. The effect of the 0, 1, or 2 parameter is implemented easily for 1 and 2 by zeroing the decimal digit array and the power array first, and using an initial power array value equal to the 1 or 2 respectively. If it is zero, the power array and the decimal digit array continue where they left off at the previous call, but with a new 10-bit value.

ADDA

To add two arrays of 10 decimal digits, ADDA adds the elements one by one while keeping track of any carry, just as it is done manually. Starting with the lowest order digits, each pair of digits are added and, if the result is less than ten, it is stored back into the first array. If it is greater than 10, it is reduced to a single digit by subtracting 10 from it and turning on carry before storing it. That carry is then added with the next higher pair of digits, and the process repeats. Carry out of the highest digits is impossible in this results software application; ADDA is coded to ignore it.

TODDAY

The NOVA RDOS operating system, like most, maintains a time of day clock and a calendar. With the appropriate system calls the hour, minute, and second are returned in separate registers, or alternatively the year, month, and day. These are all small binary numbers. ADD10 can be used to convert them to decimal, and adding "0" converts these to ASCII. CPRINT can then print each digit, plus the appropriate colons and slashes as delimiters.

USE

USE is a convenience routine, which is separated from the mainline only to make that code more readable. USE checks the flags returned by INTRO which tell which measurements are relevant and which are not. If the next measurement is irrelevant, USE returns indirectly through the address given as a calling parameter; if relevant, USE returns to the next instruction after the call.

BLOCK

To print the buffer contents in an 8 x 32 array format, BLOCK calls SET832 to initialize the formatting, DOTS to handle empty positions, and OCTDS-OCT2 to handle real values. As such, the code of BLOCK contains only branches and subroutine calls. SET832 is called first to initialize the column and row counters. DOTS is then called, which repeatedly prints "......" until a real data value is seen. If it is never seen, it returns to BLOCK which itself returns; otherwise, DOTS returns to a call to OCT2, one of the two entry points in OCTDS-OCT2. OCT2 will then print the buffer value as an octal number, and handle the "+", "D", and "S" bits. OCT2 then returns and BLOCK returns if there are no further values to be read; otherwise, OCTDS is called repeatedly to get more data values and handle them. Eventually, OCTDS will run out of values and it will return to a BLOCK return. From this description, it is clear that BLOCK does nothing but call other routines which do the work; they even decide when and if BLOCK should return or call the next routine.

DOTS

To differentiate real buffer data values from empty locations, DOTS prints "......" instead of "000000". Rather than returning and then being called again with the next value, DOTS performs read operations with the HPMD hardware itself, and does not return until real data values are seen or 256 empty locations have come and gone.

Two DIA instructions get the 20 bit buffer value shown in Figure 7. The "INFO" bit is ON only if the other 19 bits define real data; DOTS tests this, and returns to the OCT2 call if it is ON. Otherwise, PRINT is called to do the "......" and DEC832 is used to decrement the column and row counter. Depending on DEC832, DOTS loops back for more DIA's or returns to BLOCK's return if the entire buffer was empty.

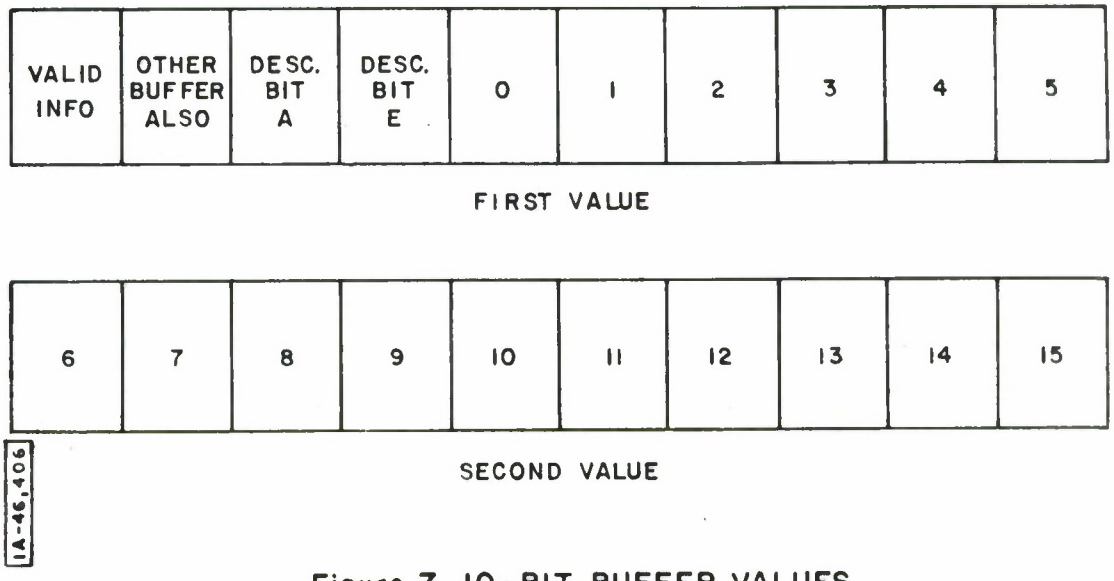

Figure 7 10-BIT BUFFER VALUES

OCTDS-OCT2

OCTDS and OCT2 are two entry points into the same routine. OCTDS performs two DIA's before falling through into the OCT2 entry, which assumes the values read are already in registers. These values are the 20 bits of a buffer entry as defined in Figure 7. OCT2 then converts the 16 bit value into a 6 digit octal number and prints it, followed by "+", "D", and "S" depending on if each symbol's corresponding bit is TRUE. DEC832 is used as it is in DOTS, to loop back for more values until the buffer is emptied.

The binary to octal conversion is done by masking only the bit or bits of interest, then converting to ASCII and using CPRINT to print it, followed by rotating the next octal digit under the mask and repeating. Additional masks test the bits which control the printing of "+", "D", or "S". If the bits are FALSE, blanks are printed to maintain the column format. DEC832 will return to one of two locations depending on whether more values remain in the buffer. These in turn cause OCTDS-OCT2 to return to one of two locations in BLOCK.

SET832

To keep track of the columns and rows while printing the 256 buffer values, SET832 and DEC832 are used. SET832 is the simpler of the two: it initializes the column counter to 8 and the row counter to 32. BLOCK calls it once per buffer printing.

DEC832

DEC832 decrements the column counter and returns if it is not zero. If it is zero, the end of the row has been reached, and three actions are taken: a carriage return and line feed are printed, the column counter is reset to 8, and the row counter is decremented. If the resulting row counter is zero, then the buffer is empty and DEC832 returns to a different location than it does if more remains. In this way, DEC832 handles the chores which must be done at the end of each line of buffer printout.

CPRINT

CPRINT uses a system call to print a single character on the line printer. It is designed for its call to be easy to insert in the calling routine by assuming the ASCII character to be printed is already in the right byte of a register. CPRINT swaps it into the left byte, and performs a Write-Sequential system call, specifying the number of bytes as one. The separation of this trivial function into a single distinct routine saved a great deal of time when the results output was altered from direct to system I/O to allow line printer spooling and printout continuity; only CPRINT and PRINT needed to be changed.

PRINT

Called by half of the routines, PRINT is an easy to use output routine for printing strings of ASCII characters. It is passed the address of such a string; first it scans the string until it sees a zero byte. This scanning allows it to count the bytes to be printed. When it has both the address and the number of bytes, a Write-Sequential system call is used to perform the actual outputting.

ERRORS

If any of several unexpected situations arise, the results software will halt. Three conditions are anticipated. $HPMD.MS may have been altered before DISPLAY is run, it may be impossible to

OPEN the line printer, and the disk may become full. $HPMD.MS is dealt with only in INTRO where attempts to position its file pointer to the measurements' flags or to the annotation may be illegal if a small file is substituted. This condition is indicated by the CPU halting with the address lights containing 1471, 1476?, or 1502. INTRO also attempts to OPEN the line printer file $LPT. If this is not possible for whatever reason, the CPU halts and the address lights will contain 1454. Pressing the "Examine AC2" switch will provide the error code in the data lights: 12 means the system can find no $LPT file, 21 implies the system thinks someone else is still using it. Finally, PRINT, CPRINT, or INTRO may halt if the disk is full and they attempt to add more to the line printer spooling buffer. In this case, the address lights will hold 1043, 1101, or 1511 and AC2 will contain 27.

It should be noted that deletion of $HPMD.MS does not cause an error; instead, INTRO returns immediately and so the printout continues without any annotation or measurements. All the possible measurements will be displayed, whether they are valid or not, since the measurement flags in $HPMD.MS are not available.

Upon an error, some degree of recovery is usually possible. An altered $HPMD.MS may result in garbage instead of annotation and measurements, but the data gathered will not be altered. Depending on the value at the flags' location, some of the data may be skipped even if valid. The "Continue" switch may be pressed to pick up execution after the halt. Almost all errors can be corrected by reloading the system and then running DISPLAY from the terminal. In some cases when one is stopped at an interrupt and wishes to continue, however, it will be necessary to choose between no printout of value or no continuation.

APPENDIX II

HARDWARE

The functions of each hardware section are defined in this appendix to convey the scope of the hardware and to show the extent that the design is dependent on the NOVA 800 and the extent it is independent.

EVENT DETECTION

The basis of the HPMD measurements is the processing of the program parameters listed in Table I. It may be noted that none of these parameters is updated each memory cycle; even Instruction and Instruction Address are available only once per machine instruction. Others, such as the I/O parameters, may be seen only rarely. It may also be noted that they consist of only three "types" of information: addresses, data, and device numbers. Every memory cycle will have both an address and a word of data, and if programmed I/O is performed, a device number will also be available. The HPMD detection logic is based on these "types" of information rather than each parameter separately, and therefore, much of the logic deals only with three sources of information, not sixteen. By sampling under the appropriate conditions, the "type" can later be categorized into the parameter of interest. For example, by sampling only on non-data-channel FETCH cycles, the addresses can be limited to only instruction addresses.

To detect parameters, it is therefore necessary for the HPMD hardware to obtain the current address, data, and device number, as well as a description of what kind of memory cycle it is. These form the basis of the conceptual, machine-independent information which flows from the interface over the cable to the rest of the HPMD hardware. Five generalized descriptor bits characterize the current memory cycle to allow discerning the parameters, and the interface synthesizes these bits from the specific machine operations. They are defined in Table VI.

Table VI

Descriptor Bits

Descriptor Bit	Definition
A	0 = CPU Cycle 1 = Data Channel Cycle
B	0 = Instruction-fetch Cycle 1 = Non-fetch Cycle
C	0 = Non-I/O Cycle 1 = I/O Execute Cycle
D	0 = Non-Interrupt Cycle 1 = Interrupt Cycle
E	0 = Load or Input Cycle 1 = Store or Output Cycle

Addresses

To obtain the address involved in each memory operation, the NOVA backplane signals must be tapped. Addresses may be specified by the CPU, in which case they appear on the MBO (Memory Buffer Output) bus, or occasionally they may be specified by a Data Channel interface and can be found on the DATA bus. A backplane signal, READIO, discerns between the two cases.

The MBO and DATA buses are connected through low-power inverters, to a 16-bit wide multiplexer. Seventy low-power inverters are used to tap all the NOVA backplane signals needed, because they require at most only a quarter the normal signal power and thus cannot significantly load down the NOVA. The multiplexer selects one of the two buses, according to the state of READIO, and sends it to a 16-bit latch. The selected bus will only have the address available for a few hundred nanoseconds, and so it must be grabbed at that moment and held in the latch. MALOAD is a backplane signal which pulses at the appropriate moment, and the latch is updated by MALOAD for each new address. This multiplexing and latching is analogous to the process the memory boards use to obtain their load or store address.

The latched address is available 300-450 nsec. before the data appears, and therefore, the old data overlaps the new address during that period. This is unacceptable if doubling is to be allowed in the detection logic, and so the address is latched again when the data appears so as to be present during the identical interval.

Data

The data value may come from the CPU, from memory, or from any I/O device. In the first two cases, it will be on the NOVA MEM bus, and for I/O it will be on the DATA bus. Descriptor bit "C", which indicates whether there is any programmed I/O during the current memory cycle, is used to select the MEM or the DATA bus with a multiplexer. The resulting 16-bit data value is latched as soon as it is known to be good by a signal called GRAB DATA. STROBE, MBLOAD, or any programmed I/O signal each imply data is or soon will be available on the appropriate bus, and so they are used to generate GRAB DATA. A 100 nsec. delay is used for STROBE and MBLOAD, while programmed I/O signals are delayed 400 nsec. This GRAB DATA triggers both the data latch and the second address latch.

Device Numbers

If the device number is the result of programmed I/O, then the DEV0-DEV5 backplane signals will carry it. In the case of an INTA (INTerrupt Acknowledge) instruction, the interrupting device puts its number on the DATA bus. As with the address and data values, it is necessary to use a multiplexer to select between two sources of information. Since device numbers are 0-77 octal, however, this path is only 6 bits wide rather than 16. Descriptor bit "D", ON only during the execute portion of INTA instructions, is used to control the multiplexer. A GRAB DEVICE pulse is synthesized by delaying INTA's and programmed I/O signals 400 nsec., and then is used to latch the device number selected by the multiplexer.

Descriptor Bits

The five descriptor bits are synthesized from NOVA memory and I/O control signals obtained from the backplane. They classify the CPU cycle as defined in Table VI. MALOAD, a pulse which tells the memory boards that an address is available, comes before each memory cycle. The HPMD generates another signal, CYCLE, by delaying MALOAD for 220 nsec., thereby getting a pulse reasonably early in each memory cycle. CYCLE is then used in synthesizing several of the descriptor bits by sampling conditions or clearing bits which may later be set. By latching the state of READIO at MALOAD time, data channel activity can be sensed. This value overlaps the previous

cycle, and so it is again sampled by CYCLE to produce descriptor bit "A", now synchronized with the rest of the bus and descriptor bit information. CYCLE is also used to latch the fetch state signal to obtain "B". CYCLE is used to clear a flip-flop which may later be set by any programmed I/O activity; this flip-flop generates descriptor bit "C". If there is such activity, it is set; if not, it remains cleared. In a similar way, bit "D" is driven by a flip-flop cleared by cycle and set by an INTA instruction. Bit "E" is the most complex, since it is based on the direction of information flow during both memory access and I/O instructions. A directly-settable, directly-clearable type D flip-flop is used to drive bit "E". At the beginning of each memory cycle, it is directly cleared by CYCLE; if at any point an output I/O instruction occurs, it is directly set. To differentiate between memory load and store operations, one need only detect a STROBE or MBLOAD pulse respectively; unfortunately, the MBLOAD pulse is also used for data input operations, and so setting bit "E" upon MBLOAD would be incorrect in some cases. By using the type D flip-flop to sample whether there is a current input operation upon each MBLOAD pulse, the "E" will be set upon MBLOAD pulses only in the absence of any input operation.

Control Pulses

In order to limit the detection operation to only the selected parameter, it is necessary to observe the proper type of information (address, data, or device number) at the proper moment. Signals called control pulses are synthesized to indicate the proper time and circumstances for each detection operation (see Figure 8).

Each control pulse must arrive at a time when the information has settled and will be correct for at least a few hundred nanoseconds to come. It also must arrive in the midst of only those memory cycles wherein the selected parameter is updated. In the simple case where instruction address (IA) is selected, the control pulse is only generated late in each CPU-initiated Fetch cycle. It must be late enough to assure the information and descriptor bits have settled down, but early enough so that there is time to perform the reduction functions.

The correct timing for all control pulses is assured by gating them with a timing signal supplied by the interface within the NOVA. A CPU timing signal, PTG3, is present only during the final quarter of each cycle; its timing is ideal for gating the control pulses.

The "proper circumstances" are more difficult to discern. This is accomplished by a two-step synthesis, in which the conditions

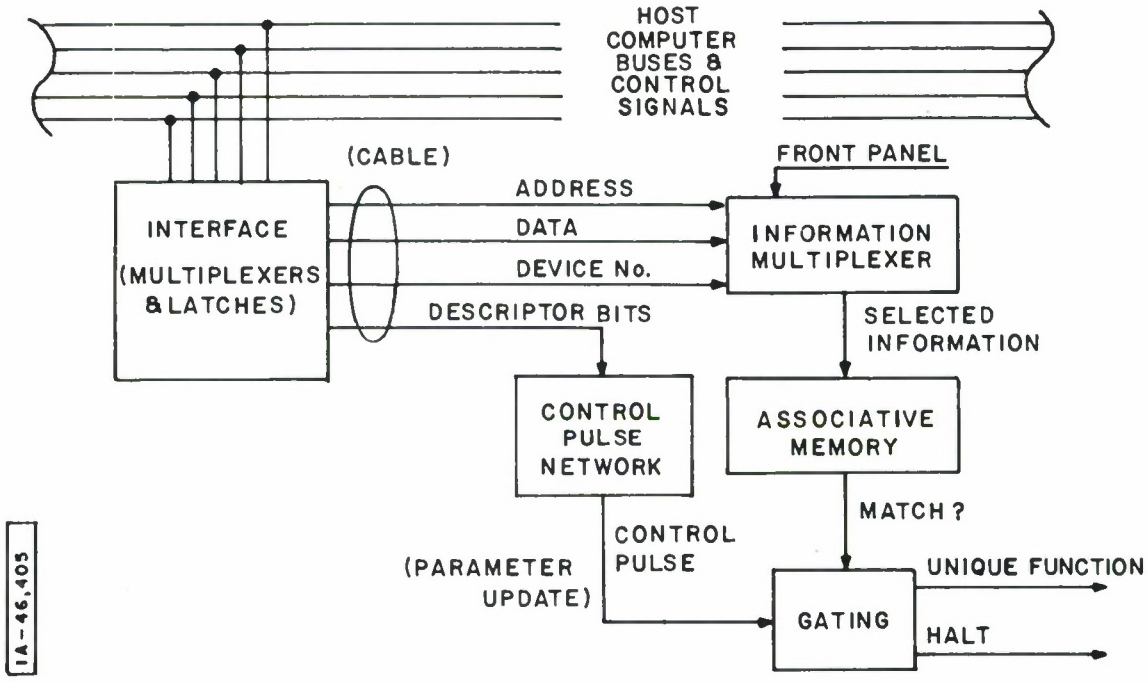

Figure 8 DETECTION LOGIC

indicating the updating of each of the sixteen parameters are detected, and then each control pulse is selected from among those sixteen signals. Table VII defines the conditions for each parameter in terms of the descriptor bit values, and these formulae are implemented in hardware by a maze of Boolean gating to produce the sixteen control pulses. Sixteen-input multiplexers select each control pulse from among these signals based on which parameter is to be detected. This selection is then gated by the timing signal to assure that the control pulse arrives at the proper time.

Table VII

Control Pulse Synthesis

Parameter	Meaning	Conditions A B C D E	Formula
INS	Instruction	0 0 0 0 0	~A~B
OPR	Operand	0 1 0 0 X	~AB~C
OPL	Operand if Load	0 1 0 0 0	~AB~C~E
OPS	Operand if Store	0 1 0 0 1	~AB~CE
DI	I/O Input Data	0 1 1 0 0	~AC~D~E
DO	I/O Output Data	0 1 1 0 1	~AC~DE
IA	Instruction Address	0 0 0 0 0	~A~B
OPA	Operand Address	0 1 0 0 X	~AB~C
OAL	Operand Address if Load	0 1 0 X 0	~AB~C~E
OAS	Operand Address if Store	0 1 0 X 1	~AB~CE
DCA	Data Channel Access Address	1 X X X X	A
AST	Address if Any Store	X 1 0 0 1	B~CE
INT	Interrupting Device #	0 1 1 1 0	~AD
DV#	I/O Device #	0 1 1 0 X	~AC~D
IN#	Input Device #	0 1 1 0 0	~AC~D~E
OU#	Output Device #	0 1 1 0 1	~AC~DE

KEY: X = don't care conditon
0 = 0 implied by other bits
1 = 1 implied by other bits

Information Multiplexer

To select the type of information to be used in the detection comparisons, a massive multiplexer is used to provide the desired address, data, or device number as requested. It is divided into ten sections, eight of which handle the information for each detection group, and the remaining two serve the buffers. Each section takes the 16 address, 16 data, and 6 device number signals provided by the NOVA interface, as well as 16 signals from the front panel switches for test purposes, and selects 16 (or 6) output signals. Since there are ten such sections in the Information Multiplexer, this is clearly one of the largest parts of the hardware.

To control each section, a two-bit value is used based on the parameter desired to select between the four sources. If the parameter is among the first six in Table I, then the type of

information needed is the current address; the next six require the current data, and the last four require the device number. To use the front panel switch register, the "Front Panel Load" switch is turned on, in which case the Information Multiplexer ignores the programmed parameter selection and provides the front panel switches value. A small set of gates can synthesize the two-bit control value from the four-bit programmed parameter and the state of its "Front Panel Load" switch. This set of gates is implemented nine times, since the Grab buffer's multiplexer is controlled directly.

Associative Memory

The heart of the detection logic is the associative memory, which performs the actual comparisons between the current information and the anticipated parameter values. It consists of 32 words of associative memory, each 16 bits wide, and is divided into 8 sections, one for each detection group. The information selected by the Information Multiplexer is fed into the corresponding associative memory group which continuously compares it with its four internal values. These comparisons proceed whether the information represents a new value of the selected parameter or not. For example, if "IA = 2000, 2001, 2002, 2003" was the event definition, then IA is the selected parameter, and 2000, 2001, 2002, and 2003 are the internal associative memory values. The associative memory, however, is fed all addresses, both of instructions and of operands, from the CPU and from the data channel. If the address, from whatever source, is among 2000-2003, then the associative memory group produces a match as its output. This process takes only 40 nsec. within the LSI chips. The timing for the entire detection process plus the buffer shift time is shown in Figure 9.

Doubling

The circuitry to implement optional detection doubling is quite simple. Sixteen programmed bits control whether doubling is required for each of the pairs of associative memory words. These bits are inverted and then OR-ed with the match outputs from words 16-31, providing signals which are TRUE if either doubling is not required or the second half of the pair is TRUE. These signals are then AND-ed with the 0-15 match outputs to effect the doubling if needed. The four matches within each group are AND-ed together to produce a single match signal, which for groups 1-4 are controlled by doubling logic, and for groups 5-8 are not. These match signals are then AND-ed with their corresponding control pulse, thereby producing eight event signals which are the electronic manifestation of the event concept developed as part of the software syntax. The

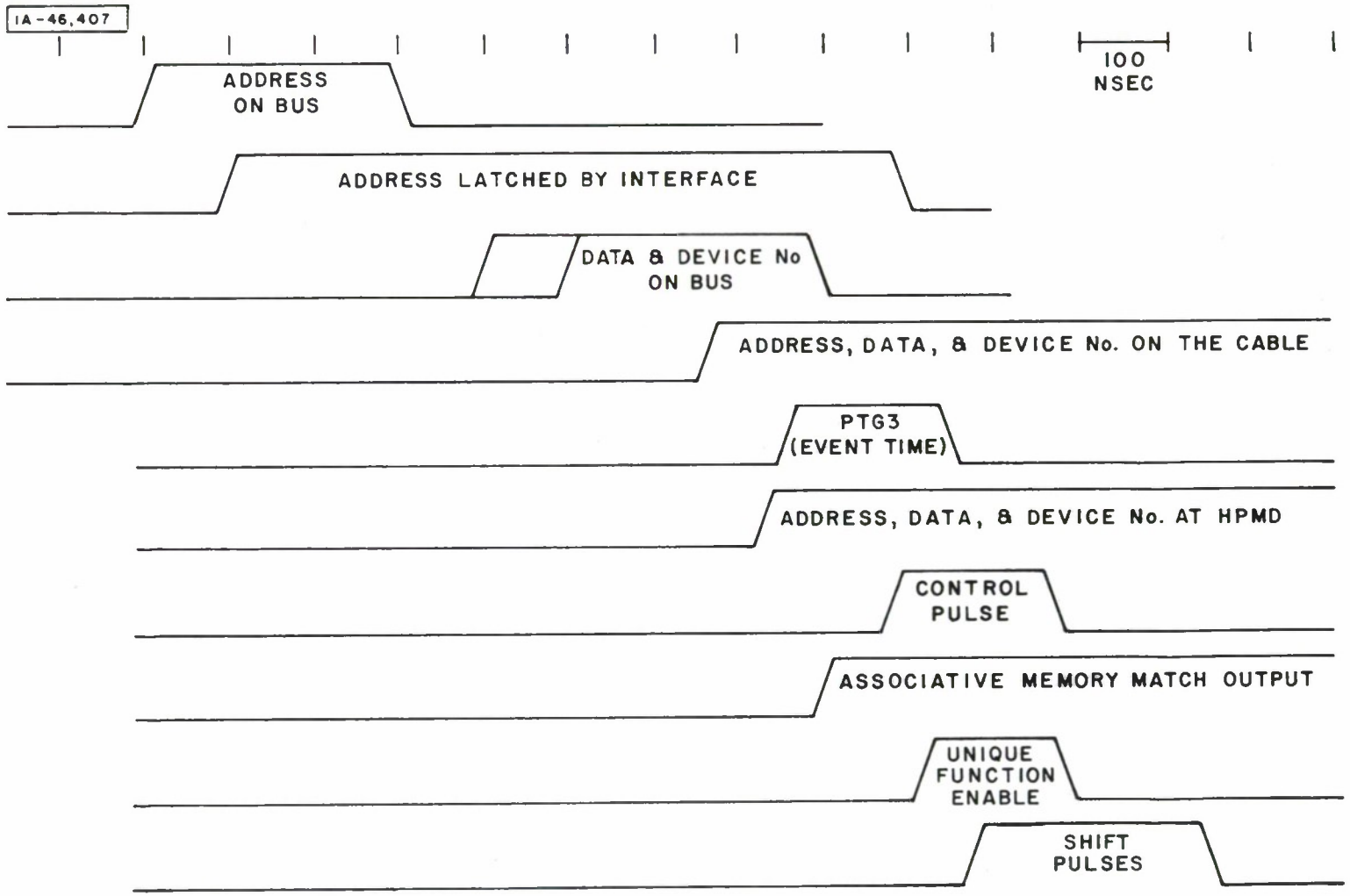

Figure 9 SIGNAL TIMING RELATIONSHIP

control pulse implies that the parameter is being updated, so if the associative memory, after any doubling gates, shows a match, then the event is detected.

INFORMATION REDUCTION

Each of the eight events can control either its Table II unique function (timer, counter, or buffer) or it can initiate a NOVA interrupt to halt the user program. Programmed "enabling" bits allow each event signal to reach the common interrupt line or its unique function input. Although the syntax does not support it, a detection group may even be programmed to do both; upon an event, the unique function could be performed as well as the program halted.

Counters

The unique functions associated with groups 5, 6, and 7 are the incrementing of counters. Initially zeroed, they increment by one upon each event detected by their detection group. Each counter is made up of five 4-bit binary counter chips connected end-to-end to produce a 20-bit counter. A common reset line zeroes all the counters at once; this line is driven by an NIOS 5 instruction in the NOVA. The 20-bit total permits a maximum count of 1,048,576 before wraparound occurs and the counter begins again from zero. The chips themselves are capable of counting at 18 MHz, which far exceeds the 1.125 MHz maximum event-detection repetition rate.

Timer

The timer is a 30-bit counter driven by a gated 10 MHz crystal oscillator. A flip-flop is set and reset by the first two detection groups respectively, and that flip-flop gates the continuously-running 10 MHz square wave. A commercially-manufactured Motorola K1091A crystal oscillator, which fits into a 14-pin DIP package, supplies the time base with a .01% practical stability. This frequency is divided by a 30-bit counter similar to the counters described in the previous section. The LSB of this counter changes state each 200 nsec., and the MSB changes each 107.3741824 sec., providing 200 nsec. resolution during a 214.7483648 second period before wraparound occurs.

Store Buffer

The store buffer obtains its parameter input as do the associative memory groups; it has its own Information Multiplexer

section and its own control pulse. A latch is used to hold the address, data, or device number selection upon the availability of a new parameter value as triggered by the control pulse.

This same control pulse initiates a buffer shift cycle. The circuitry which determines whether a shift cycle occurs or not is complex and is based on several factors. Most important, a flip-flop is set and reset by the group #3 and group #4 events as detected; group #3 enables the store buffers, and group #4 disables it. That same flip-flop is also set and reset by DIB and DIC instructions to the HPMD which allow the software to set up its initial state. Other buffer options may also suppress shift cycles. To allow a buffer control pulse to get through to the buffer itself, three conditions must be met: the store buffer must be enabled, not permanently shut down, and not frozen. The permanent shutdown flip-flop is controlled by both the disabling operation and the programmed "shutdown permanent" bit. That bit is sampled upon disabling by group #4, and if it is ON, no further shifting is allowed. Control pulses which pass the enable-disable and permanent shutdown hurdles must still be gated by the shift counter circuitry. These control pulses are fed into a divide-by-256 counter, the output of which indicates that the buffer is full. This output is used to sample the programmable bit which dictates whether the buffer should freeze when it is full. If the sampled bit is ON, then all subsequent control pulses are blocked from shifting the buffer.

If the buffer is not disabled, not permanently shut down, nor frozen, then the buffer control pulse triggers a 300 nsec. monostable multivibrator. It is the 300 nsec. pulse which is used to drive the shift registers.

The buffers are implemented out of MOS dual 256-position shift register chips, arranged as a 20 bit wide, 256 position long buffer. The input is the latched buffer input information (16 bits) plus four other bits (see Figure 7). These other bits define whether the position contains valid information, whether the other buffer is shifting on the same memory cycle, and also include the data channel and data direction descriptor bits ("A" and "E"). The valid information bit is needed since the act of reading data out of the buffer feeds meaningless data in the other end.

Grab Buffer

The grab buffer is in most ways identical to the store buffer, differing only in its source of information and triggering mechanism. Rather than being fed information in the same way as

each detection group, the grab buffer's Information Multiplexer selects between the current address, data, or device number, as well as the most recent instruction address. This selection is based on two programmable bits, not bits decoded from a selected parameter. Furthermore, a control pulse is not generated based on any parameter updating; instead, detection group #8's events trigger grab buffer shifting directly. Two other minor differences involve the grab buffer programmable options, which can inhibit the shift pulses when the store buffer is disabled or permanently shut down, but cannot permanently shut down the grab buffer.

In all other aspects, the two buffers work in the same manner. Both buffers can freeze when full if desired, and use the 20-bit wide shift registers driven by 300 nsec. pulses. The extra four bits are the same, except the "other buffer currently shifting" bit, of course, refers to the opposite buffer in each case.

Halts

The halting function can be initiated by any of 11 different causes, and so it is implemented by a single HALT signal to which any number of open-collector gates may be connected. Each detection group has such a gate, which forces the HALT line TRUE upon detecting an event if the halt-enable programmed bit for that group is TRUE. Such gates are used in a similar manner driven by each "buffer full" signal in combination with each respective "halt upon buffer full" option bit. Finally, the eleventh such open-collector gate is pulsed by the front panel "Interrupt" button, for use in both hardware and software checkout. The HALT line is sent over the cable to the NOVA interface, where the interrupt generator stops the user program.

Interrupt Generator

The interrupt generator performs seven distinct functions involved in proper interrupt handling for the HPMD on the NOVA. First of all, an enable/disable flag is maintained based on several items. It is necessary to insure that interrupts are never generated when they are not intended, and so three conditions must be met before interrupts are enabled: the cables must be connected, the "DISABLE" switch on the interface board must not be ON, and the software must have deliberately tried to enable the interrupt generation. The cable connection is sensed by sending the HPMD device's +5 volt power supply voltage over the cable; if the device is not connected and turned on, all interrupts will be suppressed. The manual switch provides another level of protection, and is useful while working on the hardware. Finally, a flip-flop is set

and reset by NIOS 5 and NIOC 5 instructions, which initiate and stop the HPMD measurements.

If enabled, the HALT signal coming over the cable is used to trigger NOVA interrupt request. The HALT signal is first latched, so an interrupt will be caused even if the HALT signal goes away. Next it is gated by the interrupt enabled signal, and then it is sampled by the periodic NOVA "REQUEST ENABLE" signal, and if ON is used to drive the common NOVA interrupt request line.

The procedure will only have an effect if the NOVA is enabled to be interrupted. To assure prompt halting despite the NOVA's interrupts being disabled, at this time the NOVA CPU interrupt enable/disable flip-flop is forced into the enable state. This is done by driving the "EXT ION EN" line, an obscure CPU signal which is rarely used. This line is held TRUE until an interrupt starts.

The fourth interrupt generator function is to handle the NOVA "INTP" lines, which are an interrupt interface daisy chain. By passing all pulses on these lines when not causing an interrupt, but blocking all pulses when it is, the interrupt generator lets the other device interfaces know to whom interrupt belongs.

To let the CPU know which interface caused the interrupt, an INTA instruction can be issued. Upon this instruction, the interrupt generator forces a "5" on the data bus where it can be read by the CPU. That "5" is the HPMD device code on the NOVA.

When the CPU interrupt enable flag is forced ON, the previous state is lost. In order to allow the software to be able to restore the original machine state for continuation after an interrupt, the previous CPU interrupt enable state must be saved. Rather than attempting to capture it at essentially the same moment as it begins to be changed, the interrupt generator keeps track of the flag state in parallel with the CPU. NIOS 77 and NIOC 77 instructions, which enable and disable the CPU interrupt capability, are used to set and reset a flip-flop which will be equivalent to the CPU flag at all times except upon an HPMD interrupt, when it will remain in its previous state while the CPU flag is forced ON.

This equivalent flag, together with the cable continuity check signal, can later be read out by the HPMD software. Bits 9 and 10 of the data word obtained by the first DIA instruction to the HPMD hardware will contain this information.

PERIPHERAL INTERFACE

Loader

The loader circuitry is responsible for the loading of all the memory elements within the hardware under the control of the software. By using an internal indexing scheme, the hardware/software protocol is reduced to a clearing pulse and a simple sequence of data transfers. A six-bit up-down counter, called the address register (AR), is used to direct incoming information to the proper storage element (see Table VIII). Initially zeroed by an NIOC 5 instruction, each subsequent DOA 0,5 instruction transfers up to 16 bits of information and then immediately increments the AR. In this way, each word goes to a different destination.

Table VIII

Loading Addresses

Address Register	Information Stored
0-31	Associative Memory Values (4 Values x 8 Groups)
32-39	Parameters for Each Group (as in Table I)
40-43	Doubling Bits, Halt and Unique Function Enable Bits, Groups 1-4
44-47	Halt and Unique Function Enable Bits, Groups 5-8
48-51	Shift Control Option Bits
52	Grab Buffer Input Selection
53-55	(unused)
56	Store Buffer Input Parameter Selection

The data word of any I/O transfer is brought to the HPMD on the data lines for detection purposes. If the I/O specifies a device number of 5 (the HPMD), then the type of I/O requested is encoded into a 3-bit field and sent over the cable. After decoding, a DOA instruction is used to store the data word in the HPMD. In this way, the same data lines which are used for carrying information for detection are also used for loading the HPMD hardware.

The AR is implemented using two 4-bit, up-down counter chips. It will be cleared by a NIOC 5 instruction and incremented by a DOA or DIA instruction (since it is also used for reading measurements back into the software). The front panel also has the capability of zeroing, incrementing, decrementing, or loading it with a specific 6-bit value.

The decoding of the AR is accomplished by a variety of arrangements depending on the nature of the corresponding storage element. The highest 3 bits are decoded into 8 lines by using part of a BCD-to-decimal decoder chip to detect AR addresses 0-7, 8-15, 16-23, etc. The first four of these lines enable four other decoders each operating on the low order 3 bits, producing 32 lines corresponding to AR addresses 0-31. These addresses belong to the 32 associative memory words, and each one is selected for loading when the AR contains its address. By zeroing the parameters to be detected, the associative memory words are fed data information, since parameter zero is "Instruction". The data information is indeed instructions during CPU Fetch cycles, but during I/O it is the I/O data word. Thus by zeroing all detection parameters, each word of associative memory will be fed all I/O data, which can then be stored into the memory when enabled by a DOA 0,5 instruction and selected by the proper AR value.

The remaining loader storage functions are far less complicated. The "32-39" high-order AR decoding line enables the parameter storage. Four 8-bit, addressable latches are all enabled by this line, and each stores one bit of each of eight parameters. The low-order 3 bits of the AR are used to steer the parameter into the proper slot in these addressable latches. In this way, the lowest 4 bits of each data word transfered during the 32nd through the 40th DOA 0,5 instruction will be stored as detection parameter values.

In a similar manner, addressable latches are used to store the eight halt enable bits, the eight reduction function enable bits, and the 2-bit value containing the grab buffer multiplexer selection as well as all buffer control option bits.

Four 4-bit latches are used to save the sixteen doubling bits, one bit for each doubling pair. Another BCD-to-decimal decoder is used to determine which latch is selected by the current AR contents. A 4-bit latch enabled by the 56-63 decoding line latches the store buffer parameter. These 4-bit and addressable 8-bit latches, together with the associative memory, constitute all of the HPMD storage.

Reading

The method used for reading the results back to the software is in many ways similar to that used by the loader. The AR is again the key to steering the information to be conveyed. Basically, successive DIA 0,5 instructions obtain the true interrupt flag, the times value, the counters' values, and the contents of both buffers. Several factors slightly complicate this method.

First of all, the path from the HPMD hardware to the NOVA interface is only 10 bits wide. This limitation was chosen since the counters and buffers are 20 bits wide and the timer is 30 bits wide, requiring a fractional-width path lest the cable requirements be unreasonable. This forces AR addresses 1-3 to be needed for the timer, 4-5, 6-7, and 8-9 for the counters, 12-13 for the store buffer, and finally 14-15 for the grab buffer. Ten and eleven remain available for future enhancements.

To allow an entire buffer to be read by successive DIA instructions, the actual AR address is altered by substituting 00110 for the highest 5 bits in the case of the store buffer and 00111 for the grab buffer; the lowest AR bit, which toggles with each DIA, was used to create an alternating pair of substitute addresses, 12-13 and 14-15. The 00110 substitution is enacted by a DIB 0,5 instruction, which transfers no meaningful data back to the NOVA, but does tell the read circuitry to begin reading the store buffer values. In a similar manner, a DIC 0,5 substitutes the grab buffer's 00111.

The AR value or its substitute will, therefore, be a number between 0-15, and a large multiplexer is used to select the proper information based on this number. This multiplexer is 10 bits wide, corresponding to the reverse data path used for reading. Inputs 0, 1, and 11 are grounded, since the interrupt generator, at the other end of the cable, will add the true interrupt flag state into 0, and 10 and 11 are unused.

This read circuitry is first used to obtain the timer and counters by an NIOC and successive DIA's. Then, a DIB and

successive DIA's read in the store buffer, and a DIC with still more DIA's get the grab buffer. Each measurement is broken into two or three 10 bit fields, and are obtained in MSB before LSB order.

CABLE

The cable is responsible for carrying many signals at very high data rates. Forty-seven signals are sent from the NOVA interface to the HPMD, and 12 proceed in the opposite direction. Ground loops, stray coupling, inter-signal coupling, and line termination distortions all had to be minimized if the information was to arrive at the other end quickly and accurately.

Ideally, a shielded twisted pair would have been available allowing each signal to be sent differentially, but the large number of signals and the limited space on the NOVA interface board ruled out so lavish a scheme. Two 50-conductor flat cables were chosen, allowing wire-wrap connectors to be bolted to the board and still remain under the .375" height limit. With 59 signals and 100 conductors, it was clear that many of them could not be sent as differential pairs. The timing signal, descriptor bits, I/O control, and HALT line were judged to be the most critical and so these were provided side-by-side conductors isolated from other pairs by grounded conductors. The timing signal, considered the most critical of all, was provided an additional grounded conductor between its differential pair. The remaining signals, the address, data, and device number buses as well as the reverse path for reading, are sent as single conductors side-by-side with each other. These signals are not as critical since they have more time to settle and all members of each bus change at once.

These bus signals have reference voltage lines interspersed among their conductors. There are four reference voltage lines for the address, data, and device number buses, and two for the reverse bus. These reference voltages are used in a differential comparison with each bus signal; in this way, some of the advantages of differential pairs, such as suppression of ground loops and stray coupling, are retained without requiring as many additional conductors.

The signals are all driven by open-collector hex inverter (or non-inverter) buffer-driver chips, which can sink 40 ma. and still develop less than +0.7 volts across their output. Using these chips for drivers, six lines could be driven by one package, saving much board real-estate at both ends of the cable.

The receivers used for all the signals were Fairchild 9615 dual differential line receivers. These work as medium-speed comparators, fast enough to handle the NOVA data rates, yet sluggish enough to ignore spikes and other very fast noise. Pull-up resistors were used on all signals, approximately matching the characteristic impedance of the flat cable (130 ohms) to minimize reflections. This resistance requires the drivers to sink 33 ma., which is an advantage in that such a low impedance and high-power level design reduces the probability that a brief and medium-energy pulse picked up from other equipment could alter the state of any signal. Because the drivers are TTL circuits, the cable voltages conform roughly to TTL requirements, with +5 and +0.7 as the actual TRUE and FALSE levels. The bus reference voltages were set to be +2.85 volts to split the difference.

FRONT PANEL

The front panel of the HPMD "box" is designed to aid in the hardware development. It allows loading all the HPMD storage, exercising the associative memory, address register, and interrupt generator, and also display of major HPMD signals. The address register is used as it would be by the software, except 16 data switches provide the information to be stored upon pushing the "Write Increment" button. Thirty-two L.E.D.s display the current output of each detection word. To obtain these DOCUMENTsignals, the associative memory outputs are latched by their corresponding control pulse. The result stays ON or OFF until another control pulse updates the display: this allows checkout of the detection functions by single-stepping the NOVA. The descriptor bits, HALT line, timer, store buffer, address register, DOA, and DIA lines are also displayed.

Push-button signals are debounced using set-reset flip-flops and then differentiated by R-C high-pass filters to produce 100 nsec. pulses. These pulses are then used to increment or decrement the AR, cause an interrupt, or reset the HPMD by emulating an NIOC 5 instruction.

The L.E.D.s are driven by the same type of hex buffer driver chips used to drive the cable. Series resistors (110 ohms) limit the current to a safe 20-30 ma level which provides adequate brightness without degrading their output very rapidly.

MECHANICAL LAYOUT

The HPMD hardware is divided between two parts: the NOVA interface and the measurement device itself. To the extent it was possible, the NOVA interface contains all the circuitry which is host-dependent.

The NOVA interface was constructed using a 15-inch square board from Data General with wire-wrap pins, to which 128 16-pin IC sockets were soldered. This board plugs directly into the NOVA chassis, occupying a slot which has been specially wired for it. Normally a memory slot, it already has available at its backplane connector all of the memory and I/O bus signals. Five CPU signals, PTG3, F, ION, PI, and EX ION EN are connected to the HPMD slot by extra backplane wiring; it is these wires which make the HPMD slot unique.

All communication with the measurement device is done over the cable, for which two 50-pin connectors have been bolted in place at one end of the board. The circuitry itself consists mainly of 14 and 16-pin I.C. chips, with a few potentiometers, discrete components, and two L.E.D.'s used in the interrupt generator checkout. Across the power supply leads at the backplane connector are a reverse diode and a zener diode to protect the NOVA from any glitches induced on these lines by the board, and vice-versa. Two 100 ufd. capacitors add protection and transient smoothing, together with sixteen 6.8 ufd. and ninety-five .05 ufd. capacitors distributed throughout the board to insure minimal power supply impedance. The power itself is derived from the NOVA, and this averages about 1.75 amps. at 5 volts.

The measurement device itself was arranged to be an efficient packaging arrangement yet provide excellent accessability, since it is a development system which required much probing and rewiring. Almost all of the circuitry was done on small cards, which could be plugged into a card cage for checkout and operation and be removed for rewiring and repair. The card cage has 13 slots, only 10 of which are needed for the HPMD as currently designed. Cable connectors are bolted on top, so their wire-wrap connections are coplanar with the card cage's backplane connections. Each slot accepts a 120-pin connector from its card, and the power supply is distributed throughout the backplane. An overvoltage protection circuit is soldered directly to the backplane which will crowbar the power supply down to 5.5 volts if a higher voltage is sensed. This both limits transients and provides protection against incorrect power supply voltages, which was most comforting when the device was run using adjustable 0-30 volt supplies early in the checkout phase.

A reverse diode also limits transients and provides additional protection. Power supply line capacitors, 50 ufd. each, are soldered to the backplane at each of the ten card slots in use. The card cage itself is mounted high in the cabinet to allow cool air to flow under the cards and be drawn upward by convection.

The circuit cards contain 20 rows of 50 wire-wrap pins each and are large enough for 60 14-pin or 50 16-pin I.C.'s. The 1000 pin matrix allows intermixing 14, 16, and 24 pin I.C.'s in a wide variety of arrangements. Additional wire-wrap pins are soldered to the edge connector, which provides access to the card cage backplane for 120 signals. Numerous .1 ufd. capacitors provide decoupling for the power lines which are distributed on both sides of each card.

The cabinet allows as much access as possible to the card cage. Both the front and back panels are hinged and latched, allowing them to be opened as doors to remove or adjust cards and to perform backplane wiring work. The front panel includes all the L.E.D.'s, switches and associated circuitry mounted directly on it so as to swing out of the way. The back panel has both the +5 and the -12 volt power supplies mounted on it, together with a fan for cooling. The top of the cabinet is perforated and is removable, allowing easy access to the cable connectors and card cage. The back panel is nearly two inches shorter than the cabinet, providing a large gap at the bottom for the cables, power cord, and circulating air to get in and out of the cabinet. This gap, together with the fan, perforated top and convection, provide exceptional cooling for both the hot power supplies and the circuit cards.